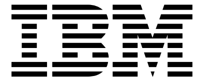
IBM® DB2® Universal Database

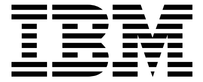


응용프로그램 개발 안내서

버전 7

SA30-0992-00

IBM® DB2® Universal Database



응용프로그램 개발 안내서

버전 7

SA30-0992-00

이 책의 정보와 지원하는 제품을 사용하기 전에 반드시 927 페이지의 『부록G. 주의사항』을 읽으십시오.

이 책에는 IBM의 특허 정보가 나와 있습니다. 이 정보는 사용권 계약하에서 제공되며, 저작권법으로 보호받습니다. 이 책에 있는 정보는 어떠한 제품도 보증하지 않으며, 이 책에 제공된 어떤 내용도 이와 같이 해석되어서는 안됩니다.

책에 대한 주문은 IBM 영업 대표나 IBM 해당 지역 사무소를 통해 하시기 바랍니다.

IBM으로 정보를 보내면, IBM은 적절한 방식으로 이를 사용하거나 배포할 수 있으며, 제공한 독자는 이에 대해 책임을 지지 않습니다.

© Copyright International Business Machines Corporation 1993, 2000. All rights reserved.

목차

제1부 DB2 응용프로그램 개발 개념 . . . 1

제1장 DB2 응용프로그램 개발 시작하기 . . . 3
이 책에 관하여 3
이 책의 사용자 4
이 책의 사용법 5
규칙 9
관련 책 10

제2장 DB2 응용프로그램 코딩 11
프로그래밍 전제조건 12
DB2 응용프로그램 코딩 개요 13
변수 선언 및 초기화 14
데이터베이스 서버에 연결 20
트랜잭션 코딩 21
프로그램 종료 23
트랜잭션 내재적 종료 24
응용프로그램 의사코드 프레임워크. 25
DB2용 응용프로그램 설계 26
데이터로 액세스. 28
데이터 값 제어 32
데이터 관계 제어 34
서버에서의 응용프로그램 논리 36
Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가 37
지원되는 SQL문 41
권한 부여 고려사항 41
동적 SQL 42
정적 SQL. 43
API 사용 43
예 44
Embedded SQL 또는 DB2 CLI 프로그램에 사용되는 데이터베이스 관리 프로그램 API . . 44
테스트 환경 설정 45
테스트 데이터베이스 작성 46
테스트 테이블 작성 46
테스트 데이터 생성 47
사용자 프로그램 수행, 테스트 및 디버깅 . . 49
SQL문 프로토타입 50

제2부 응용프로그램에서의 Embedding SQL 53

제3장 Embedded SQL 개요 55
호스트 언어에서의 Embedding SQL문 . . . 55
소스 파일 작성 및 준비 57
Embedded SQL용 패키지 작성 60
사전 처리 컴파일 60
컴파일 및 링크 64
바인딩 65
지연 바인딩의 이점 69
DB2 바인드 파일 설명 유틸리티 - db2bfd 69
응용프로그램, 바인드 파일 및 패키지 관계 70
시간소인 71
리바인딩 72

제4장 정적 SQL 프로그램 작성 75
정적 SQL 사용 특성 및 이유 76
정적 SQL의 이점 76
예: 정적 SQL 프로그램 77
정적 프로그램 작업 방법. 78
C 예: STATIC.SQC 80
Java 예: Static.sqlj 81
COBOL 예: STATIC.SQB. 82
데이터 검색 및 조작용 SQL문 코딩 83
데이터 검색 83
호스트 변수 사용 84
선언 생성기 - db2dclgn 86

© Copyright IBM Corp. 1993, 2000

표시기 변수 사용 89
데이터 유형 91
STATIC 프로그램의 표시기 변수 사용 . . 94
커서를 이용한 다중 행 선택. 95
커서 사용 및 선언 96
커서 및 작업 단위(UOW) 고려사항 . . . 97
예: 커서 프로그램 99
검색된 데이터 갱신 및 삭제 107
검색된 데이터 갱신 107
검색된 데이터 삭제 107
커서 유형 108
예: OPENFTCH 프로그램. 109
고급 화면이동 기술 118
이미 검색된 데이터의 화면이동 118
데이터 사본 보관. 118
데이터 두 번째 검색. 119
테이블 끝에 위치 설정 121
이전에 검색된 데이터 갱신. 121
예: UPDAT 프로그램 122
진단 조절 및 SQLCA 구조 133
리턴 코드 133
SQLCODE 및 SQLSTATE 133
SQLCA 구조에서의 토큰 절단 134
WHENEVER문 사용시 오류 처리 . . . 135
예외, 신호, 인터럽트 핸들러 고려사항 136
exit 목록 루틴 고려사항 137
예제 프로그램에서 GET ERROR MESSAGE 사용 137

제5장 동적 SQL 프로그램 작성 145
동적 SQL 사용 이유 145
동적 SQL 지원 명령문 146
동적 SQL과 정적 SQL 비교 147
PREPARE, DESCRIBE, FETCH 및 SQLDA 사용 150
커서 선언 및 사용 151
예: 동적 SQL 프로그램 152
SQLDA 선언 162
최소 SQLDA 구조를 사용하는 명령문 준비 163
충분한 SQLVAR 항목으로 SQLDA 할당 165
SELECT문 설명 166
행 보유를 위한 저장영역 취득 166
커서 처리 167
SQLDA 구조 할당 167
SQLDA 구조를 사용한 데이터 전달 . . 171
대화식 SQL문 처리 172
일반 사용자로부터 SQL 요청 저장 . . . 174
예: ADHOC 프로그램 174
동적 SQL로의 변수 입력 180
매개변수 표시문자 사용. 180
예: VARINP 프로그램 181
DB2 Call Level Interface(CLI) 189
Embedded SQL 및 DB2 CLI 비교 . . 189
DB2 CLI 사용시 이점 191
Embedded SQL 또는 DB2 CLI에 대한 결정 193

제6장 공통 DB2 응용프로그램 기술 . . . 195
컬럼 생성 196
식별 컬럼 196
선언된 임시 테이블 197
저장점이 있는 트랜잭션 제어 200
저장점 제한사항 201
저장점 및 DDL(Data Definition Language) 202
저장점 및 버퍼 삽입. 203
커서 블로킹과 함께 저장점 사용 203
저장점 및 XA 호환 트랜잭션 관리 프로그램 204

제3부 저장 프로시듀어 205

제7장 저장 프로시듀어 207
저장 프로시듀어 개요 207
저장 프로시듀어의 이점. 208
저장 프로시듀어 작성 211

클라이언트 응용프로그램 213
서버에서의 저장 프로시듀어 214
OLE 자동화 저장 프로시듀어 작성 . . . 234
예제 OUT 매개변수 저장 프로시듀어 . . 235
코드 페이지 고려사항 247
C++ 고려사항 248
그래픽 호스트 변수 고려사항 248
다중사이트 갱신 고려사항 249
NOT FENCED 저장 프로시듀어 249
저장 프로시듀어로부터 결과 세트 리턴. . . 251
예제: 저장 프로시듀어로부터 결과 세트 리턴 252
문제점 해결. 260

제8장 SQL 프로시듀어 작성 263
SQL 프로시듀어 및 외부 프로시듀어 비교 264
유효한 SQL 프로시듀어 본문 명령문 . . . 265
CREATE PROCEDURE문 실행 267
SQL 프로시듀어에서 조건 조절 268
조건 핸들러 선언 268
SIGNAL 및 RESIGNAL문 271
SQL 프로시듀어의 SQLCODE 및 SQLSTATE 변수 271
SQL 프로시듀어에서 동적 SQL 사용 . . . 272
중첩 SQL 프로시듀어 274
중첩 SQL 프로시듀어간의 매개변수 전달 274
중첩 SQL 프로시듀어에서 결과 새트 리턴 274
중첩 SQL 프로시듀어에 제한사항 . . . 275
SQL 프로시듀어로부터 결과 세트 리턴 . . 276
클라이언트 및 호출자 결과 세트 리턴 276
호출자 결과 세트 리턴 277
SQL 프로시듀어 디버깅 279
SQL 프로시듀어 오류 메시지 표시 . . . 279
중간 파일을 사용한 SQL 프로시듀어 디버깅 282
SQL 프로시듀어의 예 283

제9장 IBM DB2 Stored Procedure Builder 289
Stored Procedure Builder란?. 289
Stored Procedure Builder 사용 이점 . . . 291
새로운 저장 프로시듀어 작성 291
기존의 저장 프로시듀어 작업 291
Stored Procedure Builder 프로젝트 작성 292
저장 프로시듀어 디버깅 292

제4부 오브젝트 관계형 프로그래밍 . . . 295

제10장 오브젝트 관계형 기능 사용 297
DB2 오브젝트 확장 사용 이유 297
DB2의 오브젝트 관계형 기능 297

제11장 사용자 정의 구별 유형 303
구별 유형을 사용하는 이유 303
구별 유형 정의 304
규정되지 않은 구별 유형 분석 305
CREATE DISTINCT TYPE 사용 예. . . 305
예: 비용 305
예: 작업 응용프로그램 306
구별 유형에서 테이블 정의. 306
예: 판매 306
예: 응용프로그램 양식 307
구별 유형 조작 307
구별 유형 조작의 예. 308
예: 상수와 구별 유형간의 비교 308
예: 구별 유형간 유형변환 309
예: 구별 유형에 관련된 포함 311
예: 구별 유형과 관련된 전래 UDF . . . 311
예: 구별 유형과 관련된 지정 312
예: 동적 SQL에서의 지정 312
예: 서로 다른 구별 유형이 관련된 지정 313
예: UNION에서의 구별 유형 사용 . . . 314

제12장 복잡한 오브젝트 사용: 사용자 정의 구조화 유형. 315
구조화 유형 개요. 316
구조화 유형 계층 작성 317
유형화 테이블에서의 오브젝트 저장 . . . 324

컬럼에서의 오브젝트 저장 327
구조화 유형의 추가 특성 328
유형화 테이블에서의 구조화 유형 사용. . . 329
유형화 테이블 작성 329
유형화 테이블에서 데이터 처리 중 . . . 333
참조 유형 사용 334
참조 유형 비교 335
뷰 유형 작성 338
사용자 정의 유형(UDT) 또는 유형 맵핑 제거 340
뷰 변경 또는 제거 341
유형화 테이블 조회 342
참조를 해제하는 조회 342
추가 조회 스펙 기술. 345
도움말 및 추가 정보. 347
구조화 유형을 컬럼 유형으로 작성 및 사용 349
컬럼에 구조화 유형 인스턴스 삽입 . . . 349
구조화 유형 컬럼으로 테이블 정의 . . . 350
구조화 유형 속성으로 유형 정의. . . . 351
구조화 유형 값이 있는 행 삽입 351
구조화 유형 값의 검색 및 수정 353
유형과 변환 연관 355
변환 그룹 지정 위치. 358
호스트 언어 프로그램에 대한 맵핑 작성: 변환 함수 359
구조화 유형 호스트 변수 작업 381

제13장 대형 오브젝트(LOB) 사용 383
LOB란? 383
대형 오브젝트(LOB) 데이터 유형(BLOB, CLOB, DBCLOB)의 이해 384
대형 오브젝트(LOB) 위치 지정자의 이해 386
예: CLOB 값에 대해 작업할 위치 지정자 사용 388
샘플 LOBLOC 프로그램 작업 방법 . . 388
C 예: LOBLOC.SQC 389
COBOL 예: LOBLOC.SQB 391
예: LOB 표현식 평가 연기 394
샘플 LOBEVAL 프로그램 작업 방법 395
C 예: LOBEVAL.SQC 396
COBOL 예: LOBEVAL.SQB 398
표시기 변수 및 LOB 위치 지정자 . . . 401
LOB 파일 참조 변수 401
예: 파일로 문서 발췌 403
샘플 LOBFILE 프로그램 작업 방법 . . 403
C 예: LOBFILE.SQC 404
COBOL 예: LOBFILE.SQB. 406
예: CLOB 컬럼에 데이터 삽입 408

제14장 사용자 정의 함수(UDFs)와 메소드 409
함수와 메소드는 무엇입니까? 409
왜 함수와 메소드를 사용합니까?. 410
UDF 및 Method 개념 413
메소드 및 함수 구현. 416
메소드 및 함수 작성. 416
메소드 및 함수 등록. 416
UDF 및 메소드 등록의 예. 417
예: 지수화 417
예: 문자열 검색 418
예: BLOB 문자열 검색. 419
예: UDT에서의 문자열 검색 419
예: UDT 매개변수가 있는 외부 함수 . . 420
예: UDT의 AVG 421
예: 카운팅 421
예: OLE 자동화 오브젝트에서의 카운팅 422
예: 문서 ID를 리턴하는 테이블 함수 . . 422
메소드 및 함수 사용. 424
함수 참조 424
함수 호출 예 425
함수의 매개변수 표시문자 사용 426
규정화된 함수 참조 사용 426
규정화되지 않은 함수 참조 사용. . . . 427
함수 참조 요약 428

제15장 사용자 정의 함수(UDF) 433
설명 433
DB2 및 UDF간의 인터페이스 435

인수를 DB2에서 UDF로 전달 435
UDF 인수 사용 요약 451
SQL 데이터 유형이 UDF로 전달되는 방법 453
32-비트 및 64-비트 플랫폼에서의 스크래치 패드 작성 464
UDF include 파일: sqludf.h 465
Java 사용자 정의 함수 작성 및 사용 . . . 466
Java UDF 코딩 466
Java UDF 실행 방법 변경 468
Java에 대한 테이블 함수 실행 모델 . . 469
OLE 자동 UDF 작성 471
OLE 자동화 UDF 작성 및 등록 . . . 472
오브젝트 인스턴스 및 스크래치 패드 고려사항 473
SQL 데이터 유형이 OLE 자동화 UDF로 전달되는 방식 474
BASIC 및 C++에서 OLE 자동화 구현 475
OLE DB 테이블 함수 479
OLE DB 테이블 함수 작성 480
완전한 행 집합 이름. 482
OLE DB 제공자의 서버 이름 정의. . . 484
사용자 맵핑 정의 484
지원되는 OLE DB 데이터 유형 485
스크래치 패드 고려사항. 488
테이블 함수 고려사항 490
테이블 함수 오류 처리 491
스칼라 함수 오류 처리 492
UDF 매개변수 또는 결과로 LOB 위치 지정자 사용 493
LOB 위치 지정자 사용에 대한 시나리오 498
기타 코딩시 고려사항 499
도움말 및 추가 정보. 499
UDF 제한사항 및 조건. 502
UDF 코드 예 505
예: 정수 나누기 연산자. 506
예: CLOB 접기, 모음 찾기 510
예: 계수기(Counter) 515
예: 날씨 테이블 함수 517
예: LOB 위치 지정자를 사용하는 함수 525
예: BASIC으로 작성한 계수기(Counter) OLE 자동 UDF 528
예: C++로 작성한 계수기(Counter) OLE 자동 UDF 530
UDF 디버깅 535

제16장 활동중인 DBMS에서 트리거 사용 537
트리거를 사용하는 이유. 537
트리거의 이점 539
트리거 개요. 539
트리거 이벤트 540
영향을 받는 행 집합. 541
트리거 수준. 542
트리거 활성화 시간 543
전이 변수 544
전이 테이블. 546
트리거 조치. 548
트리거 조치 조건 548
트리거 SQL문. 549
SQL 트리거 명령문 내의 함수 549
트리거 연쇄. 551
참조 제한조건과의 상호작용 551
복수 트리거 정렬 551
트리거, 제한조건, UDT, UDF 및 LOB간의 시너지 553
정보 추출 553
테이블에 대한 조작 금지 554
비지니스 규칙 정의 554
조치 정의 555

제5부 DB2 프로그래밍 고려사항 . . . 557

제17장 복잡한 환경에서의 프로그래밍 . . . 559
자국어 지원 고려사항 560
조합 순서 개요 560
코드 페이지 값 구하기 566
응용프로그램에서 로케일 구하기 567

자국어 지원 응용프로그램 개발 568
DBCS 문자 집합. 577
확장 UNIX 코드(EUC) 문자 세트 . . . 578
DBCS 환경에서 CLI/ODBC/JDBC/SQLJ 프로그램 수행 579
일본어 및 대만어 EUC 및 UCS-2 코드 세트 변환 581
다중 사이트 갱신시 고려사항 597
원격 작업 단위(RUOW) 597
다중 사이트 갱신. 598
Host 또는 AS/400 서버에 액세스 606
다중 스레드 데이터베이스 액세스 607
다중 스레드 사용시 권장사항 608
코드 페이지 및 국가 코드에 대하여 작업하는 다중 스레드 UNIX 응용프로그램. . 609
다중 스레드 사용시 잠재적 함정. . . . 610
동시 트랜잭션 612
동시 트랜잭션 사용시 잠재적 함정 . . . 613
X/Open XA 인터페이스 프로그래밍 고려사항 615
응용프로그램 링크 619
네트워크에 걸쳐 있는 대용량의 데이터에 대한 작업 619

제18장 파티션된 환경에서의 프로그래밍 고려사항 621
성능 향상 621
FOR READ ONLY 커서 사용 621
방향이 지정된 DSS와 지역 바이패스 사용 621
버퍼 삽입 사용 623
예: 대용량의 데이터 추출(largevol.c) . . 630
테스트 환경 작성 636
오류 처리시 고려사항 636
심각한 오류. 637
병합된 복수 SQLCA 구조. 637
오류를 리턴하는 파티션 식별 639
디버깅 639
루핑 또는 일시정지한 응용프로그램 진단 639

제19장 DB2 연합 시스템용 프로그램 작성 641
DB2 연합 시스템 소개 641
데이터 소스 테이블 및 뷰 액세스 642
별명에 대한 작업. 642
분리 레벨을 사용하여 데이터 무결성 유지보수 647
데이터 유형 맵핑에 대한 작업 647
DB2에서 지역으로 정의할 데이터 유형을 판별하는 방법 648
기본 데이터 유형 맵핑 648
기본 유형 맵핑을 겹쳐쓰고 새로운 맵핑을 작성하는 방법 649
분산 요청(DR)을 사용하여 데이터 소스 조회 650
분산 요청(DR) 코딩 650
서버 옵션을 사용하여 최적화 촉진 . . . 652
데이터 소스 함수 호출 653
데이터 소스 함수를 호출하도록 DB2 설정 653
함수 호출의 오버헤드 감소. 654
CREATE FUNCTION MAPPING문에서 함수 이름 지정 656
함수 맵핑 분리 656
Pass-Through를 사용하여 데이터 소스 직접 조회 656
Pass-Through 세션으로 SQL 처리 . . . 656
고려사항 및 제한사항 657

제6부 언어 고려사항 659

제20장 C 및 C++ 프로그래밍 661
C 및 C++에 대한 프로그래밍 고려사항 . . 661
C 및 C++의 언어 제한사항 661
C 및 C++의 Trigraph 절차 662
C++ 유형 장식물 고려사항 662
C 및 C++에 대한 입력과 출력 파일 . . . 663
C 및 C++에 대한 파일. 663
C 및 C++에 파일 포함. 667
C 및 C++에 SQL문 포함. 668
C 및 C++의 호스트 변수 670

C 및 C++의 호스트 변수 명명 670
C 및 C++의 호스트 변수 선언 671
C 및 C++의 표시기 변수 676
C 또는 C++의 그래픽 호스트 변수 선언 676
C 또는 C++의 LOB 데이터 선언 . . . 679
C 또는 C++의 LOB LOB 위치 지정자 선언 682
C 또는 C++의 파일 참조 선언 683
C 및 C++의 호스트 변수 초기화 . . . 684
C 매크로 예 684
C 및 C++의 호스트 구조 지원 685
C 및 C++의 표시기 테이블 687
C 및 C++의 널로 종료된 문자열 . . . 689
C 및 C++의 포인터 데이터 유형 . . . 690
C 및 C++에서 클래스 데이터 구성원을 호스트 변수로 사용 692
C 및 C++에서 구성 및 구성원 연산자 사용 693
C 및 C++에서의 그래픽 호스트 변수 처리 694
C 및 C++의 일본어 또는 대만어 EUC 및 UCS-2 고려사항 700
C 및 C++의 지원되는 SQL 데이터 유형 701
C 및 C++의 2진 데이터용 708
C 및 C++의 SQLSTATE 및 SQLCODE 변수 708

제21장 Java 프로그래밍 709
Java에 대한 프로그래밍 고려사항 709
SQLJ와 JDBC 비교. 710
다른 언어를 능가하는 Java의 이점 . . . 710
Java에 있는 SQL 보안. 711
Java에 대한 소스 및 출력 파일 711
Java 클래스 라이브러리. 711
Java 패키지 712
Java에서 지원되는 SQL 데이터 유형 . . 712
Java에 있는 SQLSTATE 및 SQLCODE 값 714
Java 추적 기능 715
Java 응용프로그램 및 애플릿 작성 . . . 715
JDBC 프로그래밍. 718
DB2Appl 프로그램의 작업 방식. . . . 719
JDBC 응용프로그램 분배 722
JDBC 애플릿 분배 및 수행 722
JDBC 2.0 723
SQLJ 프로그래밍. 726
DB2 SQLJ 지원 727
Java에 SQL문 포함 730
Java 호스트 변수. 736
SQLJ에 있는 저장 프로시듀어 및 함수에 대한 호출 736
SQLJ 프로그램 컴파일 및 수행 737
SQLJ 변환기 옵션 739
Java에 있는 저장 프로시듀어 및 UDF . . 740
Java 클래스를 넣을 위치 741
루틴에서 Java 클래스 갱신 742
Java에서 저장 프로시듀어 디버깅 . . . 743
Java 저장 프로시듀어 및 UDF 746
JDBC 1.22에 LOB 및 그래픽 오브젝트 사용 750
JDBC 및 SQLJ 상호운용성 752
세션 공유 752
Java의 연결 자원 관리 752

제22장 Perl 프로그래밍 755
Perl 프로그래밍 고려사항 755
Perl 제한사항 755
Perl을 사용한 데이터베이스 연결 756
Perl에서 결과 페치 756
Perl의 매개변수 표시문자 757
Perl의 SQLSTATE 및 SQLCODE 변수 758
Perl DB2 응용프로그램 예 758

제23장 COBOL 프로그래밍 761
COBOL에 대한 프로그래밍 고려사항 . . . 761
COBOL에서 언어 제한사항 761
COBOL에 대한 입력과 출력 파일 761

COBOL의 Include 파일 762
COBOL에서의 Embedding SQL문. 765
COBOL에서 호스트 변수 768
COBOL에서 오스트 변수 명명 768
호스트 변수 선언 769
COBOL에서 표시기 변수 773
COBOL에서 LOB 선언 773
COBOL에서 LOB LOB 위치 지정자 선언 774
COBOL에서 파일 참조 선언 775
COBOL에서 호스트 구조 유지 776
COBOL에서 표시기 테이블 778
COBOL 그룹 데이터 항목에서 REDEFINES 사용 779
BINARY/COMP-4 COBOL 데이터 유형 사용 780
COBOL에서 지원되는 SQL 데이터 유형 780
COBOL에서 FOR BIT DATA 784
COBOL에서 SQLSTATE 및 SQLCODE 변수 785
COBOL에 대한 일본어 또는 대만어 EUC 및 UCS-2 고려사항 785
오브젝트 지향 COBOL. 786

제24장 FORTRAN 프로그래밍 787
FORTRAN에 대한 프로그래밍 고려사항 787
FORTRAN에서 언어 제한사항 787
FORTRAN에서의 참조에 의한 호출 . . 788
FORTRAN에서 디버깅 및 주석 행. . . 788
FORTRAN에 대한 프로그래밍 고려사항 788
FORTRAN에 대한 입력과 출력 파일 . . . 788
FORTRAN에 대한 포함 파일 789
FORTRAN에 파일 포함 792
FORTRAN에서의 SQL문 포함 793
FORTRAN에서 호스트 변수 794
FORTRAN에서의 호스트 변수 명명 . . 795
호스트 변수 선언 795
FORTRAN에서 표시기 변수 798
FORTRAN에서 LOB 선언 798
FORTRAN에서 LOB 위치 지정자 선언 799
FORTRAN에서 파일 참조 선언 800
FORTRAN에서 지원되는 SQL 데이터 유형 800
FORTAN에서 SQLSTATE 및 SQLCODE 변수 803
FORTRAN에서의 다중 바이트 문자 세트에 대한 고려사항 803
FORTRAN에 대한 일본어 또는 대만어 EUC 및 UCS-2 고려사항 803

제25장 REXX 프로그래밍. 805
REXX에 대한 프로그래밍 고려사항. . . . 805
REXX에 대한 언어 제한사항. 806
REXX에서 SQLEXEC, SQLDBS 및 SQLDB2 등록 806
REXX에서 Embedding SQL문 808
REXX에서 호스트 변수 810
REXX에서 호스트 변수 명명. 810
REXX에서 호스트 변수 조회. 810
REXX의 표시기 변수 811
사전 정의된 REXX 변수 811
REXX의 LOB 호스트 변수 813
REXX에서 LOB 위치 지정자 선언. . . 814
REXX에서 LOB 파일 참조 선언 . . . 815
REXX에서 LOB 호스트 변수 지움. . . 816
REXX에서 지원되는 SQL 데이터 유형 . . 817
REXX에서 커서 사용 819
REXX에 대한 실행 요구사항. 819
REXX에 대한 바인드 파일 820
REXX용 API 구문 821
REXX저장 프로시듀어 823
REXX에서 저장 프로시듀어 호출 . . . 823
REXX에 대한 일본어 또는 대만어 EUC 고려사항 826

제7부 부록 및 끝머리 827

부록A. 지원되는 SQL문 829

부록B. 샘플 프로그램 835
Embedded SQL 샘플이 없는 DB2 API 840
DB2 API Embedded SQL 샘플 844
DB2 API가 없는 Embedded SQL 샘플 846
사용자 정의 함수 샘플 847
DB2 Call Level Interface 샘플. 848
Java 샘플 849
SQL 프로시듀어 샘플 851
ADO, RDO 및 MTS 샘플 853
오브젝트 링크 및 샘플 포함 854
명령행 처리기 샘플 855
로그 관리 User Exit 샘플. 856

부록C. DB2DARI 및 DB2GENERAL 저장 프로시듀어와 UDF 857
DB2DARI 저장 프로시듀어 857
클라이언트 응용프로그램에서 SQLDA 사용 858
DB2DARI 클라이언트에서 호스트 변수 사용 858
저장 프로시듀어에서 SQLDA 사용. . . . 859
데이터 구조 사용의 요약 860
입력/출력 SQLDA 및 SQLCA 구조 . . 861
DB2DARI 저장 프로시듀어에 대한 저장 값 862
DB2GENERAL UDF 및 저장 프로시듀어 863
지원되는 SQL 데이터 유형 863
저장 프로시듀어 및 UDF용 클래스. . . 865
비분리 저장 프로시듀어. 872
예제 입력-SQLDA 프로그램 873
입력-SQLDA 클라이언트 응용프로그램의 작업 방식 874
C 예: V5SPCLI.SQC 876
예제 입력-SQLDA 저장 프로시듀어의 작업 방식 878
C 예: V5SPSRV.SQC 879

부록D. 호스트 또는 AS/400 환경에서 프로그래밍 881
DDL(Data Definition Language) 사용 . . 883
DML(Data Manipulation Language) 사용 883
숫자 데이터 유형 884
혼합 바이트 데이터 884
긴 필드 884
대형 오브젝트(LOB) 데이터 유형 . . . 884
사용자 정의 유형(UDT) 885
ROWID 데이터 유형 885
64비트 정수(BIGINT) 데이터 유형 . . . 885
DCL(Data Control Language) 사용 . . . 885
연결 및 연결해제 885
사전 처리 컴파일 886
블로킹 886
패키지 속성. 887
C 널로 종료되는 문자열 888
독립형 SQLCODE 및 SQLSTATE . . 889
정렬 순서 정의. 889
참조 무결성 관리 890
잠금 890
SQLCODE와 SQLSTATE 사이의 차이점 891
시스템 카탈로그 사용 891
검색 할당에서 숫자 변환 오버플로우 . . . 891
분리 레벨 892
저장 프로시듀어 893
Stored Procedure Builder. 895
최소단위 복합 SQL이 아님 896
DB2 Connect와 다중 사이트 갱신 897
DB2 Connect에서 지원하는 호스트 또는 AS/400 서버 SQL문 898
DB2 Connect에서 거부하는 호스트 또는 AS/400 서버 SQL문 899

부록E. EBCDIC 2진 조합 시뮬레이트 . . 901

부록F. DB2 라이브러리 사용. 907
DB2 PDF 파일 및 인쇄된 책 907
DB2 정보 907
PDF 책 인쇄 917
인쇄된 책 주문 917

DB2 온라인 문서. 919
온라인 도움말 액세스 919
정보 온라인 보기 921
DB2 마법사 사용. 923
문서 서버 설정 924
정보 온라인 검색 925

부록G. 주의사항 927

등록 상표 930

색인 933

IBM에 문의 967
제품 정보 967

제1부 DB2 응용프로그램 개발 개념

© Copyright IBM Corp. 1993, 2000

제1장 DB2 응용프로그램 개발 시작하기

이 책에 관하여 3
이 책의 사용자 4
이 책의 사용법 5
규칙 9
관련 책 10

이 책에 관하여

이 책에서는 DB2 데이터베이스에 액세스하는 응용프로그램의 설계 및 코드화 방법에 대해 설명합니다. 지원되는 호스트 언어 프로그램에서 SQL을 사용하는 것에 대한 상세한 정보가 제공됩니다. 특정 운영 체제에 대한 언어 지원에 대한 자세한 내용은 *응용프로그램 빌드 안내서*을 참조하십시오. 이 책은 DB2 응용프로그램을 작성할 때 도움을 줄 수 있는 일부 DB2 유틸리티의 개요도 제공합니다. 이런 유틸리티에는 37 페이지의 『Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가』 및 289 페이지의 『제9장 IBM DB2 Stored Procedure Builder』가 포함됩니다.

다음과 같이 데이터에 액세스할 수 있습니다.

- Java용 Embedded SQL(SQLJ)을 포함하여 호스트 언어로 내포된 SQL문
- Java Database Connectivity(JDBC), Perl DBI 및 DB2 Call Level Interface(DB2 CLI)를 포함한 동적 API

이 책에서는 *CLI Guide and Reference*에 논의된 DB2 CLI를 제외한 데이터에 액세스하는 모든 방법에 대해 설명합니다. JDBC, SQLJ 및 DB2 CLI는 Embedded SQL을 통해 사용할 수 없는 일부 데이터 액세스 기능을 제공합니다. 이러한 기능에는 여러 결과 세트를 리턴하는 스크롤 가능 커서 및 저장 프로시듀어가 포함됩니다. 어떤 데이터 액세스 방법을 사용할 것인지 결정하려면 28 페이지의 『데이터로 액세스』에 있는 설명을 보십시오.

DB2 응용프로그램의 설계, 작성 및 테스트를 위해 이 책에 있는 정보를 효과적으로 사용하려면 이 책과 함께 *SQL 참조서*의 내용을 참조해야 합니다. DB2 데이터베이스에 액세스하기 위해 응용프로그램에 있는 DB2 CLI(Call Level Interface) 또는 ODBC(Open Database Connectivity) 인터페이스를 사용할 경우, *CLI Guide*

© Copyright IBM Corp. 1993, 2000

*and Reference*의 내용을 참조하십시오. 사용자 응용프로그램의 DB2 관리 API를 사용하여 데이터베이스 관리 프로그램 관리 함수를 수행하려면 *Administrative API Reference*의 내용을 참조하십시오.

응용프로그램 중 한 부분은 클라이언트에서, 다른 부분은 서버에서 수행되는 응용 프로그램을 개발할 수도 있습니다. 버전 7의 DB2에서 저장 프로시듀어에 대해 플랫폼을 통해 확장된 이식성 및 확장성(scalability)을 지원합니다. 저장 프로시듀어에 대해서는 207 페이지의 『제7장 저장 프로시듀어』에 설명되어 있습니다.

DB2에 대한 오브젝트 기반 확장을 사용하여 DB2 응용프로그램을 기존의 DB2 응용프로그램보다 강력하고, 융통성있고, 활동성있도록 만들 수 있습니다. 확장에는 대형 오브젝트(*LOBs*), 구별 유형, 구조화된 유형, 사용자 정의 함수(*UDF*) 그리고 트리거가 있습니다. 이러한 DB2 기능에 대해서는 다음에서 설명됩니다.

- 297 페이지의 『제10장 오브젝트 관계형 기능 사용』
- 303 페이지의 『제11장 사용자 정의 구별 유형』
- 315 페이지의 『제12장 복잡한 오브젝트 사용: 사용자 정의 구조화 유형』
- 383 페이지의 『제13장 대형 오브젝트(LOB) 사용』
- 409 페이지의 『제14장 사용자 정의 함수(UDFs)와 메소드』
- 433 페이지의 『제15장 사용자 정의 함수(UDF)』
- 537 페이지의 『제16장 활동중인 DBMS에서 트리거 사용』

이 책에서 DB2란 UNIX, Linux, OS/2 그리고 Windows 32-비트 운영 체제에서의 DB2 Universal Database 제품을 의미합니다. 다른 플랫폼에서의 DB2는 AS/400용 DB2 Universal Database와 같은 특정 제품 이름과 플랫폼을 사용합니다.

이 책의 사용자

이 책은 SQL 사용에 익숙하며, 지원되는 프로그래밍 언어가 여러 개인 프로그래머를 대상으로 합니다.

이 책의 사용법

이 책은 타스크별로 다음과 같이 구성되어 있습니다.

- 제1부 DB2 응용프로그램 개발 개념에는 이 책을 사용하는 데 필요한 정보와, DB2 Universal Database용 응용프로그램을 개발하기 위해 사용할 수 있는 방법에 대한 개요가 들어 있습니다.
  - 제1장 DB2 응용프로그램 개발 시작하기에는 이 책의 구조와 그 안에서 사용된 규약에 대해 설명합니다.
  - 제2장 DB2 응용프로그램 코딩에서는 DB2를 사용하는 전반적인 응용프로그램 개발 프로세스를 소개합니다. 여기에서는 응용프로그램을 코드화하기 전에 고려해야 할 중요한 응용프로그램 설계 문제에 대해 논의 및 비교를 합니다. 이 장에는 응용프로그램 개발을 시작할 수 있는 테스트 환경 설정을 돕기 위한 정보가 있습니다.
- 제2부 응용프로그램에서의 Embedding SQL에는 정적 및 동적 SQL을 응용프로그램에 내포시키는 방법이 나와 있습니다. 이 정보에는 Embedded SQL 응용프로그램 작성을 돕기 위해 사용할 수 있는 유틸리티에 대한 설명이 포함됩니다.
  - 호스트 언어에서의 Embedding SQL문은 C/C++, Java 그리고 COBOL과 같은 호스트 언어에서 SQL을 포함시켜서 DB2 응용프로그램을 작성하는 프로세스를 설명합니다. 여기에는 DB2 사전 처리 컴파일러, 응용프로그램 컴파일 및 링크 그리고 데이터베이스 Embedded SQL문을 바인드하는 작업의 개요가 있습니다.
  - 제4장 정적 SQL 프로그램 작성에는 정적 SQL문을 사용하여 DB2 Embedded SQL 응용프로그램을 코드화하는 것에 대한 세부사항이 있습니다. 또한, 정적 SQL 사용에 대한 자세한 안내 및 유의사항이 들어 있습니다.
  - 제5장 동적 SQL 프로그램 작성에서는 동적 SQL문을 사용한 DB2 Embedded SQL 응용프로그램을 코드화하는 것에 대한 세부사항이 있습니다. 또한, 동적 SQL을 사용하는 것에 대한 상세한 안내 및 유의사항이 들어 있습니다.

  - 제6장 공통 DB2 응용프로그램 기술은 일반적인 응용프로그램 개발 문제에 도움을 주는 DB2 기능을 설명합니다. 이런 기능에는 고유 행 식별자를 자동으로 작성하여 표현식에서 동적으로 파생되는 컬럼을 작성하고 선언된 임시 테이블을 작성하고 사용하는 기능이 포함됩니다.
- 제3부 저장 프로시듀어에서는 클라이언트/서버 환경에서 수행되는 데이터베이스 응용프로그램의 성능을 향상시키기 위해 저장 프로시듀어를 사용하는 방법에 대해 설명합니다.
  - 제7장 저장 프로시듀어는 저장 프로시듀어와 호스트 언어를 사용하여 저장 프로시듀어를 호출하는 클라이언트 응용프로그램을 작성하는 방법을 설명합니다.
  - 제8장 SQL 프로시듀어 작성은 CREATE PROCEDURE문을 실행하여 SQL에서 저장 프로시듀어를 작성하는 방법을 설명합니다. SQL 프로시듀어는 CREATE PROCEDURE문의 본문에서 SQL을 사용하여 절차적 논리를 인코드합니다.
  - 제9장 IBM DB2 Stored Procedure Builder는 DB2용 저장 프로시듀어의 신속한 개발을 지원하는 그래픽 응용프로그램인 IBM DB2 Stored Procedure Builder를 설명합니다. Stored Procedure Builder는 SQL과 Java 저장 프로시듀어를 작성하는 데 도움을 줍니다.
- 제4부 오브젝트 관계형 프로그래밍에서는 DB2가 제공하는 오브젝트 관계형 지원을 사용하는 방법에 대해 설명합니다. 이 정보에는 대형 오브젝트(LOB), 사용자 정의 함수, 사용자 정의 구별 유형 및 트리거 사용 방법에 대한 소개 및 상세한 안내가 포함됩니다.
  - 제10장 오브젝트 관계형 기능 사용에서는 DB2의 오브젝트 지향 기능을 소개합니다. 오브젝트 지향 문맥에서 대형 오브젝트(LOB), 사용자 정의 함수 및 사용자 정의 구별 유형과 같은 DB2 기능을 이용하는 응용프로그램으로 기존의 응용프로그램을 확장시키는 방법에 대해 설명합니다.
  - 제11장 사용자 정의 구별 유형에서는 응용프로그램에서 사용자 자신의 데이터 유형을 작성 및 사용하는 방법에 대해 설명합니다. 이는 내장 데이터 유형에 대한 오브젝트 지향 확장자에 대한 기초로서 구별 유형을 사용하는 방법에 대해 설명합니다.

- 제12장 복잡한 오브젝트 사용: 사용자 정의 구조화 유형은 응용프로그램에서 구조화 유형을 작성하고 사용하는 방법을 설명합니다. 이는 오브젝트를 구조화 유형의 계층으로 모델화하고 구조화 유형의 인스턴스를 테이블에서 행 또는 컬럼으로 액세스하며 구조화 유형을 응용프로그램 내외에서 바인드하는 방법을 설명합니다.
- 제13장 대형 오브젝트(LOB) 사용에서는 최고 2GB의 2진 또는 텍스트 문자열로서 데이터 오브젝트를 저장할 수 있는 데이터 유형을 정의 및 사용하는 방법에 대해 설명합니다. 또한, 네트워크 환경에서 LOB를 효율적으로 사용하는 방법에 대해서도 설명합니다.
- 제14장 사용자 정의 함수(UDFs)와 메소드에서는 SQL로의 사용자 자신의 확장자 작성 방법에 대해 설명합니다. 데이터 오브젝트 작업을 표현하기 위해 UDF를 사용하는 방법에 대해 설명합니다.
- 제15장 사용자 정의 함수(UDF)에서는 DB2 응용프로그램을 확장시키는 사용자 정의 함수 작성 방법에 대해 설명합니다. 주제에는 사용자 정의 함수(UDF) 작성에 대한 세부사항, 사용자 정의 함수(UDF)에 대한 프로그래밍 유의사항, 이 중요 기능을 구현하는 방법을 보여 주는 몇 가지 예가 포함됩니다. 또한, 이 장에서는 사용자 정의 테이블 함수, OLE DB 테이블 함수, OLE 자동화 UDF에 대해 설명합니다.
- 제16장 활동중인 DBMS에서 트리거 사용에서는 모든 데이터베이스 응용프로그램 내에서 비지니스 규칙을 캡슐화 및 강화시키기 위해 트리거를 사용하는 방법에 대해 설명합니다.

• 제5부 DB2 프로그래밍 고려사항에는 특수 응용프로그램 개발 유의사항에 대한 정보가 있습니다.

- 제17장 복잡한 환경에서의 프로그래밍에서는 자국어 지원, 데이터베이스 및 응용프로그램에 대한 Extended UNIX® Code(EUC) 코드 페이지, 작업 단위(UOW) 내에서의 복수 데이터베이스 액세스, 멀티스레드 응용프로그램 작성과 같은 고급 프로그래밍 주제 항목을 다룹니다.
- 제18장 파티션된 환경에서의 프로그래밍 고려사항에서는 파티션된 환경에서 수행되는 응용프로그램을 개발중인 경우의 프로그래밍 유의사항에 대해 설명합니다.

- 제19장 DB2 연합 시스템용 프로그램 작성에서는 연합 서버를 통해 DB2 계열 및 Oracle 데이터 소스로부터 투명하게 데이터에 액세스하는 응용프로그램 작성 방법에 대해 설명합니다.

• 제6부 언어 고려사항에서는 DB2가 지원하는 프로그래밍 언어에 대한 구체적인 정보가 제공됩니다.

- 제20장 C 및 C++ 프로그래밍에서는 C 및 C++로 작성된 데이터베이스 응용프로그램에 관한 호스트 언어별 정보에 대해 설명합니다.
- 제21장 Java 프로그래밍에서는 JDBC 또는 SQLJ를 사용하여 Java로 작성된 데이터베이스 응용프로그램에 관한 호스트 언어별 정보에 대해 설명합니다.
- 제22장 Perl 프로그래밍은 Perl 데이터베이스 인터페이스(DBI) 모듈에 대해 DBD::DB2 데이터베이스 드라이버를 사용하여 Perl로 작성된 데이터베이스 응용프로그램에 관한 호스트 언어 고유 정보를 설명합니다.
- 제23장 COBOL 프로그래밍에서는 COBOL로 작성된 데이터베이스 응용프로그램에 관한 호스트 언어별 정보에 대해 설명합니다.
- 제24장 FORTRAN 프로그래밍에서는 FORTRAN으로 작성된 데이터베이스 응용프로그램에 관한 호스트 언어별 정보에 대해 설명합니다.
- 제25장 REXX 프로그래밍에서는 REXX로 작성된 데이터베이스 응용프로그램에 관한 호스트 언어별 정보에 대해 설명합니다.

• 부록에는 DB2 응용프로그램 개발시 참조해야 할 보충 설명이 들어 있습니다.

- 부록A. 지원되는 SQL문에는 DB2 Universal Database에 의해 제공되는 SQL문이 나열되어 있습니다.
- 부록B. 샘플 프로그램에는 지원되는 호스트 언어에 대해 제공되는 샘플 프로그램에 대한 정보가 제공되고, 그 작동 방식에 대해 설명합니다.
- 부록C. DB2DARI 및 DB2GENERAL 저장 프로시듀어와 UDF에는 이전 버전의 DB2 Universal Database와 호환되는 UDF 및 저장 프로시듀어를 작성할 때 사용할 수 있는 정보가 제공됩니다.
- 부록D. 호스트 또는 AS/400 환경에서 프로그래밍에서는 분산 환경에서 사용자의 응용프로그램에 있는 AS/400 데이터베이스 서버 또는 호스트에 액세스할 경우 DB2 Connect에 대한 프로그래밍 유의사항이 있습니다.

– 부록E. EBCDIC 2진 조합 시뮬레이트에서는 EBCDIC 또는 사용자가 정의한 조합 순서에 따라 DB2 문자열을 조합하는 방법에 대해 설명합니다.
– 부록F. DB2 라이브러리 사용에는 DB2 Universal Database 제품에 대한 자세한 정보를 얻을 수 있는 곳이 나와 있습니다.

규칙

이 책에서는 다음과 같은 규약을 사용합니다.

디렉토리 및 경로

이 책에서는 디렉토리 분리용 UNIX 규약을 사용합니다(예: sqllib/samples/java). /를 \로 변경하고 해당 설치 드라이브 및 디렉토리를 사전 보류함으로써 Windows 32비트 운영 체제 및 OS/2 경로로 이들 경로를 변환할 수 있습니다.

이탤릭체

다음 중 하나를 나타냅니다.

- 새 용어 소개
- 사용자가 제공하는 값 또는 변수 이름
- 정보의 또다른 출처(예. 책 또는 CD-ROM)
- 일반 강조

UPPERCASE

다음 중 하나를 나타냅니다.

- 약어
- 데이터베이스 관리 프로그램 데이터 유형
- SQL문

예

다음 중 하나를 나타냅니다.

- 코드화 예 및 코드 부분
- 시스템에 의해 표시되는 것과 유사한 출력 예
- 구체적인 데이터 값의 예
- 시스템 메시지 예
- 파일 및 디렉토리 이름
- 입력할 정보

- Java 메소드 이름
- 함수 이름
- API 이름

볼드체 볼드체는 포인트를 강조표시합니다.

관련 책

다음 안내서에서는 특정 국가 및 국제용 응용프로그램 개발 방법에 대해 설명합니다.

문서 번호	책 제목
SE09-8001-03	*National Language Design Guide, Volume 1*
SE09-8002-03	*NLS Reference Manual, Release 4*

제2장 DB2 응용프로그램 코딩

프로그래밍 전제조건 12
DB2 응용프로그램 코딩 개요 13
변수 선언 및 초기화 14
데이터베이스 관리 프로그램과 상호작용 하는 변수 선언 14
오류 및 경고 조절 17
추가 비실행 명령문 사용 20
데이터베이스 서버에 연결 20
트랜잭션 코딩 21
트랜잭션 시작 22
트랜잭션 종료 22
프로그램 종료 23
트랜잭션 내재적 종료 24
대부분의 지원 운영 체제에서 24
Windows 32비트 운영 체제 25
DB2 Context API 사용시 25
응용프로그램 의사코드 프레임워크 25
DB2용 응용프로그램 설계 26
데이터로 액세스 28
Embedded SQL 28
DB2 Call Level Interface(DB2 CLI) 및 Open Database Connectivity(ODBC) 30
JDBC 30
Microsoft 스펙 31
Perl DBI 31
Query Products 31
데이터 값 제어 32
데이터 유형 32
고유 제한조건 32
테이블 점검 제한조건 33
참조 무결성 제한조건 33
점검 옵션 보기 33
응용프로그램 논리 및 프로그램 변수 유형 34
데이터 관계 제어 34
참조 무결성 제한조건 34
트리거 35
응용프로그램 논리 35
서버에서의 응용프로그램 논리 36
저장 프로시듀어 36
사용자 정의 함수 36
트리거 37
Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가 . . . 37
Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가 활성화 40
Microsoft Visual C++에 대한 IBM DB2 Universal Database 도구 추가 활성화 40
지원되는 SQL문 41
권한 부여 고려사항 41
동적 SQL 42
정적 SQL 43
API 사용 43
예 44
Embedded SQL 또는 DB2 CLI 프로그램에 사용되는 데이터베이스 관리 프로그램 API . . 44
테스트 환경 설정 45
테스트 데이터베이스 작성 46
테스트 테이블 작성 46
테스트 데이터 생성 47
사용자 프로그램 수행, 테스트 및 디버깅 . . 49
SQL문 프로토타입 50

© Copyright IBM Corp. 1993, 2000

프로그래밍 전제조건

이 장에서는 DB2 응용프로그램의 논리적 부분 모델이 제시되어 있으며 지원되는 DB2 프로그래밍 API의 각 장점에 대해 논의합니다. DB2 응용프로그램 개발에 익숙치 못한 프로그래머는 이 장 전체를 숙독해야 합니다.

이 책의 응용프로그램 개발 과정에서는 사용자가 적절한 운영 환경을 설치했다는 것을 전제로 합니다. 즉, 다음 항목들이 제대로 설치 및구성되어 있다고 전제합니다.

- 지원 컴파일러 또는 응용프로그램 개발용 해석기
- DB2 Universal Database, 지역 또는 원격
- DB2 응용프로그램 개발 클라이언트.

이 타스크의 수행 방법에 대한 세부사항은 사용자 운영 환경에 대한 *응용프로그램 빌드 안내서* 및 *빠른 시작* 책을 참조하십시오.

DB2 응용프로그램 개발 클라이언트(DB2 응용프로그램 개발 클라이언트)가 설치된 클라이언트나 서버에서 응용프로그램을 *개발*할 수 있습니다. 서버, DB2 Run-Time Client 또는 DB2 Administrative Client에서 응용프로그램을 *수행*할 수 있습니다. 또한, 클라이언트를 설치할 때 "Java 지원" 구성요소를 설치하면 이들 클라이언트에서 Java JDBC 프로그램도 개발할 수 있습니다. 이는 이들 클라이언트에서 DB2 응용프로그램을 실행할 수 있음을 의미합니다. 그러나, 이들 클라이언트와 함께 DB2 응용프로그램 개발 클라이언트도 설치하지 않으면, 이 클라이언트에서는 JDBC 응용프로그램만 개발할 수 있습니다.

DB2는 사전처리 컴파일러를 통해 C, C++, Java (SQLJ), COBOL 및 FORTRAN 프로그래밍 언어를 지원합니다. 또한, DB2는 Perl, Java (JDBC) 및 REXX 동적 해석 언어에 대한 지원을 제공합니다. DB2가 제공하는 특정 사전처리 컴파일러 및 사용자 플랫폼에서 지원되는 언어에 대해서는 *응용프로그램 빌드 안내서*에서 자세한 내용을 참조하십시오.

주: 미래에 DB2 버전 5에서 안정화된 FORTRAN 및 REXX 지원이 계획되어 있지만, FORTRAN 또는 REXX 지원에 대한 추가 처리는 계획되어 있지 않습니다.

DB2는 제공되는 샘플 프로그램을 수행할 때 필요한 샘플 데이터베이스를 제공합니다. 샘플 데이터베이스 및 목차를 보려면 *SQL 참조서*에서 자세한 내용을 참조하십시오.

DB2 응용프로그램 코딩 개요

DB2 응용프로그램은 여러 개의 부분들로 구성되어 있습니다.

1. 변수 선언 및 *초기화*
2. 데이터베이스에 연결
3. 하나 이상의 *트랜잭션* 수행
4. 데이터베이스로부터 연결해제
5. 프로그램 종료

*트랜잭션*이란 데이터베이스에 확약하기 전에 성공적으로 종료되어야 하는 데이터베이스 조작 세트를 말합니다. Embedded SQL과 함께 트랜잭션은 암시적으로 시작되며, 응용프로그램이 COMMIT문 또는 ROLLBACK문을 실행할 때 종료됩니다. 트랜잭션의 예로는 고객의 예금 입력 및 잔고 갱신을 들 수 있습니다.

호스트 언어로부터 embedded SQL문으로의 전이를 처리하려면 일부 SQL문은 프로그램의 시작 및 끝에 나타나야 합니다.

모든 프로그램 시작 부분에는 다음 항목들이 포함되어야 합니다.

- 데이터베이스 관리 프로그램이 호스트 프로그램과 상호작용하는 모든 변수 및 데이터 구조의 선언문
- SQL 통신 영역(SQLCA) 설정에 의한 오류 조절을 위해 제공하는 SQL문

  Java로 작성된 DB2 응용프로그램은 `SQLException`을 드로우하며, 이는 SQLCA를 사용하지 않고 `catch` 블록에서 처리합니다.

모든 프로그램의 본문에는 데이터를 액세스 및 관리하는 SQL문이 들어 있습니다. 이들 명령문은 트랜잭션을 구성합니다. 트랜잭션에는 다음 명령문이 들어 있어야 합니다.

- 데이터베이스 서버로의 연결을 설정하는 CONNECT문
- 다음 명령문이 둘 이상 포함합니다.
  - 데이터 조작 명령문(예를 들어, SELECT문)

– 데이터 정의 명령문(예를 들어, CREATE문)

– 데이터 제어 명령문(예를 들어, GRANT문)

- 트랜잭션 종료를 위한 COMMIT문 또는 ROLLBACK문

일반적으로 응용프로그램의 끝 부분에는 다음과 같은 SQL문이 포함됩니다.

- 데이터베이스 서버로의 프로그램 연결 해제
- 모든 자원 제거

변수 선언 및 초기화

DB2 응용프로그램을 코드화하려면 우선 다음을 선언해야 합니다.

- 데이터베이스 관리 프로그램과 상호작용하는 변수
- SQLCA(적용가능한 경우)

데이터베이스 관리 프로그램과 상호작용하는 변수 선언

데이터베이스 관리 프로그램과 상호작용하는 모든 변수는 SQL 선언 절에서 선언되어야 합니다. 다음과 같은 구조로 SQL 선언 절을 코드화해야 합니다.

1. SQL문 `BEGIN DECLARE SECTION`
2. 하나 이상의 변수 선언문으로 구성된 그룹
3. SQL문 `END DECLARE SECTION`

SQL declare 부분에 선언되어 있는 호스트 프로그램 변수를 *호스트 변수*라고 합니다. SQL문의 *호스트-변수* 참조에서 호스트 변수를 사용할 수 있습니다. *호스트-변수*는 *SQL 참조서*의 구문 도표에 사용되는 태그입니다. 한 프로그램에 여러 개의 SQL declare 부분이 포함될 수 있습니다.

각 호스트 변수의 속성은 SQL문에서 변수가 사용되는 방식에 따라 달라집니다. 예를 들어, DB2 테이블에 데이터를 저장하거나 데이터를 받는 변수의 데이터 유형 및 길이는 액세스 중인 컬럼과 호환되는 것과 같아야 합니다. 각 변수의 데이터 유형을 알아보려면 DB2 데이터 유형을 잘 알고 있어야 하는데, 이에 대해서는 91 페이지의 『데이터 유형』에 설명되어 있습니다.

SQL 오브젝트를 나타내는 변수 선언: DB2 버전 7의 경우, 테이블, 별명, 뷰 및 상관 이름의 최대 길이는 128바이트입니다. 컬럼 이름의 최대 길이는 30바이트입니다. DB2 버전 7에서 스키마 이름의 최대 길이는 30바이트입니다. 차후 DB2

릴리스에서는 컬럼 이름 및 SQL 오브젝트의 기타 식별자 이름을 최고 128바이트까지 증가시킬 수 있습니다. 128바이트가 안되는 SQL 오브젝트를 나타내는 변수를 선언할 경우 차후 SQL 오브젝트 식별 길이가 증가되면 응용프로그램의 안정성에 영향을 줄 수 있습니다. 예를 들어, 스키마 이름을 보유하도록 char[9]*schema_name* 변수를 C++ 응용프로그램에서 선언할경우, 최대 길이가 8바이트인 DB2 버전 6에 허용되는 스키마 이름에 대해 사용자 응용프로그램이 제대로 작동됩니다.

```
char[9] schema_name; /* holds null-delimited schema name of up to 8 bytes;
 works for DB2 Version 6, but may truncate schema names in future releases */
```

그러나, 최대 길이가 30바이트인 스키마 이름을 허용하는 DB2 버전 7로 데이터베이스를 이주시킬 경우, 스키마 이름 LONGSCHEMA1과 LONGSCHEMA2가 구분되지 않게됩니다. 데이터베이스 관리 프로그램은 이 스키마 이름들을 8바이트로 제한하여 LONGSCHE로 되므로, 스키마 이름을 구별해야 하는 응용프로그램에서의 명령문은 실패하게 됩니다. 응용프로그램의 길이를 증가시키려면 다음과 같이 128바이트 길이를 가진 스키마 이름 변수를 선언하십시오.

```
char[129] schema_name; /* holds null-delimited schema name of up to 128 bytes
                          good for DB2 Version 7 and beyond */
```

차후의 응용프로그램 조작을 향상시키려면, 응용프로그램에 길이가 128바이트인 SQL 오브젝트 이름을 나타내는 모든 변수들을 선언해 보십시오. 이보다 긴 변수의 경우 더 많은 시스템 자원이 필요하다는 것에 비해 호환성이 향상된다는 이점이 있다는 것을 고려해야 합니다.

이러한 코딩 조작을 쉽게 하고 C/C++ 응용프로그램 코드를 보다 명료하게 하려면 이러한 SQL 오브젝트 식별자 길이를 선언할 때 C 매크로확장자를 사용해 보십시오. sql.h 포함 파일은 SQL_MAX_IDENT를 128로 선언하므로, SQL_MAX_IDENT 매크로를 이용하여 SQL 오브젝트 식별자를 쉽게 선언할 수 있습니다. 예를 들면 다음과 같습니다.

```
#include <sql.h>
    char[SQL_MAX_IDENT+1] schema_name;
    char[SQL_MAX_IDENT+1] table_name;
    char[SQL_MAX_IDENT+1] employee_column;
    char[SQL_MAX_IDENT+1] manager_column;
```

C 매크로 확장에 대한 세부사항은 684 페이지의 『C 매크로 예』의 내용을 참조하십시오.

호스트 변수를 SQL문에 연관시키기: 호스트 변수를 사용하여 데이터베이스 관리 프로그램으로부터 데이터를 받거나, 호스트 프로그램으로부터 데이터베이스 관리 프로그램으로 데이터를 전송할 수 있습니다. 데이터베이스 관리 프로그램으로부터 데이터를 받는 호스트 변수를 *출력 호스트 변수*라 하고, 호스트 프로그램으로부터 데이터베이스 관리 프로그램으로 데이터를 전송하는 호스트 변수를 *입력 호스트 변수*라 합니다.

다음 SELECT INTO문을 보십시오.

```
SELECT HIREDATE, EDLEVEL
  INTO :hdate, :lvl
  FROM EMPLOYEE
  WHERE EMPNO = :idno
```

여기에는 두 개의 출력 호스트 변수, `hdate` 및 `lvl`과, 하나의 입력 호스트 변수인 `idno`가 포함됩니다. EMPLOYEE 테이블로부터 검색되는 행의 EMPNO를 결정하기 위해 데이터베이스 관리 프로그램은 `idno` 호스트 변수에 저장된 데이터를 사용합니다. 데이터베이스 관리 프로그램이 검색 기준에 맞는 행을 찾으면 `hdate` 및 `lvl`은 각각 HIREDATE 및 EDLEVEL 컬럼에 저장된 데이터를 받습니다. 이 명령문은 EMPLOYEE 테이블의 컬럼을 사용하여 호스트 프로그램과 데이터베이스 관리 프로그램 사이의 상호작용을 보여줍니다.

테이블의 각 컬럼에는 CREATE TABLE 정의에 있는 데이터 유형이 지정됩니다. 이 문서에서 각 언어별 해당 장의 *지원 SQL 데이터 유형* 부분에 정의된 호스트 언어 데이터 유형과 이 데이터 유형을 연관시켜야 합니다. 예를 들어, INTEGER 데이터 유형은 32비트의 기호화된 정수입니다. 이는 각 호스트 언어로 된 다음 데이터 설명 항목과 같습니다.

C/C++:

```
sqlint32 variable_name;
```

Java:

```
int variable_name;
```

COBOL:

```
01  variable-name  PICTURE S9(9) COMPUTATIONAL-5.
```

FORTRAN:

```
INTEGER*4 variable_name
```

지원 SQL 데이터 유형 및 해당 호스트 언어 데이터 유형 목록에 대해서는 다음을 참조하십시오.

- C/C++용, 701 페이지의 『C 및 C++의 지원되는 SQL 데이터 유형』
- Java용, 712 페이지의 『Java에서 지원되는 SQL 데이터 유형』
- COBOL용, 780 페이지의 『COBOL에서 지원되는 SQL 데이터 유형』
- FORTRAN용, 800 페이지의 『FORTRAN에서 지원되는 SQL 데이터 유형』
- REXX용, 817 페이지의 『REXX에서 지원되는 SQL 데이터 유형』

컬럼에 사용할 호스트 변수의 정의 방법을 결정하려면 해당 컬럼의 SQL 데이터 유형을 알아야 합니다. 데이터베이스에 작성된 모든 테이블에 대한 정보가 들어 있는 뷰 세트인 시스템 카탈로그를 조회해 보십시오. 이 카탈로그에 대해서는 *SQL 참조서*에 설명되어 있습니다.

데이터 유형을 알아냈으면 호스트 언어 장의 변환표를 참조하여 해당 선언문을 코드화할 수 있습니다. Declaration Generator 유틸리티(db2dclgn)를 사용하여 데이터베이스에 제공된 테이블에 대한 해당 선언문을 생성할 수도 있습니다. db2dclgn에 대한 세부사항은 86 페이지의 『선언 생성기 - db2dclgn』 및 *Command Reference*의 내용을 참조하십시오.

87 페이지의 표4에는 지원 호스트 언어에 있는 선언문 예가 나와 있습니다. REXX 응용프로그램은 LOB 위치 지정자 및 파일 참조 변수를 제외하고, 호스트 변수를 선언할 필요가 없습니다. 변수 목차를 통해 런타임시에 기타 호스트 변수 데이터 유형 및 크기를 알아낼 수 있습니다.

표4에는 BEGIN문 및 END DECLARE SECTION문도 있습니다. SQL문의 분리문자가 언어에 따라 어떻게 다른지 보십시오. 이러한 명령문들의 정확한 위치 지정, 지속 및 분리 규칙에 대해서는 이 책의 언어별 해당 장을 참조하십시오.

오류 및 경고 조절

SQL 통신 영역(SQLCA)에 대한 자세한 내용은 이 장의 뒷부분에서 다룰 것입니다. 이 절에서는 간단한 개요만 제시됩니다. SQLCA를 선언하려면 프로그램에서 INCLUDE SQLCA문을 코드화하십시오.

C 또는 C++ 응용프로그램용:

```
EXEC SQL INCLUDE SQLCA;
```

Java 응용프로그램의 경우: Java에서는 SQLCA를 명시적으로 사용하지 않습니다. 대신, SQLException 인스턴스 메소드를 사용하여 SQLSTATE 및 SQLCODE 값을 얻습니다. 714 페이지의『Java에 있는 SQLSTATE 및 SQLCODE 값』에서 자세한 내용을 참조하십시오.

COBOL 응용프로그램용:

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

FORTRAN 응용프로그램용:

```
EXEC SQL INCLUDE SQLCA
```

프로그램을 사전 처리할 때 데이터베이스 관리 프로그램은 호스트 언어 변수 선언문을 INCLUDE SQLCA문 위치에 삽입합니다. 시스템은 경고 플래그, 오류 코드 및 진단 정보용 변수를 사용하여 사용자 프로그램과 통신합니다.

시스템은 각 SQL문을 실행한 후 SQLCODE 및 SQLSTATE 모두에서 리턴 코드를 리턴합니다. SQLCODE는 명령문 실행을 요약ㅎ는 정수 값이고, SQLSTATE는 IBM의 관계형 데이터베이스 제품을 통해 일반적인 오류 코드를 제공하는 문자 필드입니다. SQLSTATE는 ISO/ANS SQL92 및 FIPS 127-2 표준을 따릅니다.

주: FIPS 127-2는 *Federal Information Processing Standards Publication 127-2 for Database Language SQL*을 의미합니다. ISO/ANS SQL92는 *American National Standard Database Language SQL X3.135-1992* 및 *International Standard ISO/IEC 9075:1992, Database Language SQL*을 말합니다.

SQLCODE가 0보다 작을 경우 오류가 발생했고 명령문이 처리되지 않았음을 나타냅니다. SQLCODE가 0보다 큰 경우 경고가 나타났지만 여전히 명령문이 처리되었음을 나타냅니다. SQLCODE 및 SQLSTATE에 대해서는 *메시지 참조서*의 내용을 참조하십시오.

각 SQL문 다음에 시스템이 오류 점검을 제어하도록 하려면 WHENEVER문을 사용하십시오.

주: Java용 Embedded SQL(SQLJ) 응용프로그램에서는 WHENEVER문을 사용할 수 없습니다. 714 페이지의 『Java에 있는 SQLSTATE 및 SQLCODE 값』에 설명되어 있는 SQLException 메소드를 사용하여 SQL문에 의해 리턴된 오류들을 처리하십시오.

다음 WHENEVER문은 SQLCODE가 음수일 때 시스템에게 실행 상황을 표시합니다.

```
WHENEVER SQLERROR GO TO errchk
```

즉, SQL 오류가 발생할 때마다 프로그램 제어는 errchk와 같은 레이블 뒤에 오는 코드로 이동됩니다. 이 코드에는 SQLCA에 있는 오류 표시기를 분석하기 위한 논리가 포함되어야 합니다. ERRCHK 정의에 따라 다음 절차의 프로그램 지시를 실행하거나 일부 특수 기능을 수행하며, 대부분의 경우에서처럼 현재 *트랜잭션*을 구간 복원하고 프로그램을 종료하는 조치가 취해집니다. 트랜잭션에 대한 세부사항은 21 페이지의 『트랜잭션 코딩』의 내용을, 사용자의 응용프로그램에서의 오류 점검 방법에 대한 세부사항은 133 페이지의 『진단 조절 및 SQLCA 구조』의 내용을 참조하십시오.

WHENEVER SQLERROR문을 사용할 때의 실행시 유의사항. 응용프로그램의 오류 조절 코드에 SQL문에 포함되어 있고, 이러한 명령문의 실행 결과 원래 오류를 처리하는 동안 오류가 발생할 경우, 무한 루프로 될 가능성이 있습니다. 이런 경우 문제를 해결하기가 어렵습니다. WHENEVER SQLERROR 목적지의 최초 명령문은 WHENEVER SQLERROR CONTINUE여야 합니다. 이 명령문은 오류 핸들러를 재설정합니다. 이 명령문 다음에 SQL문을 안전하게 사용할 수 있습니다.

C++ 또는 C로 작성된 DB2 응용프로그램의 경우, 응용프로그램이 여러 개의 소스 파일로 구성되어 있으면 SQLCA의 다중 정의를 피하려면 이 파일 중 하나에 EXEC SQL INCLUDE SQLCA문이 포함되어야 합니다. 나머지 소스 파일은 다음 행을 사용해야 합니다.

```
#include "sqlca.h"
extern struct sqlca sqlca;
```

응용프로그램이 ISO/ANS SQL92 또는 FIPS 127-2 표준을 따라야 하는 경우, 위 명령문이나 INCLUDE SQLCA문을 사용하지 마십시오. ISO/ANS SQL92 및 FIPS 127-2 표준에 대한 세부사항은 18 페이지의 『FIPS 127-2 및 ISO/ANS SQL92 정의』의 내용을 보십시오. 위 명령문을 코드화하는 대안에 대해서는 다음을 참조하십시오.

- C 또는 C++ 응용프로그램의 경우, 708 페이지의 『C 및 C++의 SQLSTATE 및 SQLCODE 변수』
- COBOL 응용프로그램의 경우, 785 페이지의 『COBOL에서 SQLSTATE 및 SQLCODE 변수』
- FORTRAN 응용프로그램의 경우, 803 페이지의 『FORTAN에서 SQLSTATE 및 SQLCODE 변수』

추가 비실행 명령문 사용

일반적으로 기타 *비실행* SQL문도 이 프로그램 절의 부분이기도 합니다. *SQL 참조서* 및 이 안내서의 후속 장 모두에서 비실행 명령문에 대해 논의합니다. 비실행 명령문의 예는 다음과 같습니다.

- INCLUDE `text-file-name`
- INCLUDE SQLDA
- DECLARE CURSOR

데이터베이스 서버에 연결

사용자 프로그램은 실행 SQL문을 수행하기 전에 목표 데이터베이스 서버로의 연결을 설정해야 합니다. 이 연결로, 프로그램을 실행하는 사용자의 권한 부여 ID와 프로그램이 실행되는 데이터베이스 서버 이름 모두가 식별됩니다. 일반적으로 사용자의 응용프로그램 프로세스는 한 번에 하나의 데이터베이스 서버로만 연결할 수 있습니다. 이러한 서버를 *현재 서버*라고 합니다. 그러나, 응용프로그램은 다중 사이트 갱신 환경내에서 여러 개의 데이터베이스 서버로 연결할 수 있습니다. 이런 경우 하나의 서버만 현재 서버가 될 수 있습니다. 다중 사이트 갱신에 대한 세부사항은 598 페이지의 『다중 사이트 갱신』의 내용을 참조하십시오.

사용자 프로그램은 데이터베이스 서버로의 연결도 설정할 수 있습니다.

- 명시적, CONNECT문 사용
- 암시적, 기본 데이터베이스 서버로 연결

- Java 응용프로그램의 경우, `Connection` 인스턴스를 통해

연결 상태에 대한 논의 및 CONNECT문 사용 방법에 대해서는 *SQL 참조서*의 내용을 참조하십시오. 초기설정시 응용프로그램 리퀘스터(AR)는 기본 데이터베이스 서버를 설정합니다. 내재된 연결이 작동되어 있는 경우, 초기설정 후 시작된 응용프로그램 프로세스가 기본 데이터베이스 서버로 내재적으로 연결합니다. 응용프로그램에 의해 실행되는 최초의 SQL문으로 CONNECT문을 사용하는 것이 좋습니다. 이렇게 하면 기본 데이터베이스가 아닌 SQL문을 실행할 우려가 없어집니다.

연결이 설정된 후에는 프로그램이 다음을 실행하는 SQL문을 발행할 수 있습니다.

- 데이터 조작
- 데이터베이스 오브젝트 정의 및 유지보수
- 사용자 권한 부여, 데이터베이스에 대한 변경사항 확약과 같은 제어 조작 초기설정

CONNECT RESET문, CONNECT TO문 또는 DISCONNECT문이 발행될 때까지 연결이 지속됩니다. 다중 사이트 갱신 환경에서, DB2 RELEASE와 DB2 COMMIT가 발행될 때까지도 연결이 지속됩니다. CONNECT TO문은 다중 사이트 갱신을 사용할 때 연결을 종료하지 않습니다(598 페이지의 『다중 사이트 갱신』 참조).

트랜잭션 코딩

트랜잭션이란 데이터베이스 관리 프로그램이 전체로 취급하는 (중재 호스트 언어 코드를 가진) 일련의 SQL문입니다. 트랜잭션을 흔히 *작업 단위(UOW)*라고도 합니다.

트랜잭션 레벨에서 데이터의 일관성을 유지하려면 트랜잭션 내의 모든 조작이 완료되거나 어떤 조작도 완료되지 *않아야* 합니다. 예를 들어, 프로그램이 한 계좌에서 돈을 인출해서 다른 계좌에 추가한다고 가정하십시오. 단일 이 두 갱신을 모두 단일 트랜잭션에 둔 상태에서 갱신이 진행되는 동안 시스템 장애가 발생할 경우, 시스템을 다시 시작할 때 데이터베이스 관리 프로그램은 트랜잭션이 시작되기 전 상태로 데이터를 자동 복원시킵니다. 프로그램 오류가 발생하면 데이터베이스 관리 프로그램은 오류가 있는 명령문에 의해 변경된 모든 변경사항을 복원합니다. 데

이터베이스 관리 프로그램은 특별히 구간 복원하지 않는 한 오류가 있는 명령문을 실행하기 전에 트랜잭션에서 수행된 작업을 실행 취소시킵니다.

단일 응용프로그램 내에서 하나 이상의 트랜잭션을 코드화할 수 있으며, 단일 트랜잭션 내부로부터 하나 이상의 데이터베이스에 액세스하는 것이 가능합니다. 하나 이상의 데이터베이스에 액세스하는 트랜잭션을 다중 사이트 갱신이라고 합니다. 이러한 주제에 대해서는 597 페이지의 『원격 작업 단위(RUOW)』 및 598 페이지의 『다중 사이트 갱신』의 내용을 참조하십시오.

트랜잭션 시작

트랜잭션은 내재적으로 최초의 *executable* SQL문으로 시작하고, COMMIT문 또는 ROLLBACK문으로, 또는 프로그램이 종료될 때 종료됩니다.

대비적으로 다음 여섯 개 문장은 실행 명령문이 아니므로 트랜잭션을 시작하지 **않습니다**.

```
BEGIN DECLARE SECTION          INCLUDE SQLCA
END DECLARE SECTION            INCLUDE SQLDA
DECLARE CURSOR                 WHENEVER
```

실행 SQL문은 항상 트랜잭션 내에서 발생합니다. 프로그램에 트랜잭션 종료 후 실행 SQL문이 포함되어 있는 경우 자동으로 새로운 트랜잭션을 시작합니다.

트랜잭션 종료

트랜잭션 종료 방법은 다음과 같이 두 가지입니다.

- COMMIT문을 이용하여 변경사항을 저장한 후 종료
- ROLLBACK문을 이용하여 변경사항을 저장하지 않은 후 종료

COMMIT문 사용: 이 명령문은 현재 트랜잭션을 종료합니다. 현재 트랜잭션 도중의 데이터베이스 변경사항을 다른 프로세스가 볼 수 있도록 합니다.

응용프로그램 여건이 허용하는 가능한 한 신속히 변경사항을 확약해야 합니다. 특히, 터미널로부터의 입력을 대기하는 동안 확약되지 않은 변경사항이 보유되지 않도록 프로그램을 작성하십시오. 이렇게 하면 데이터베이스 자원을 오랫동안 보유할 수 있습니다. 이렇게 자원을 보유함으로써 이들 자원을 필요로 하는 다른 응용프로그램이 수행되지 못하도록 합니다.

COMMIT문은 호스트 변수 내용에는 영향을 미치지 않습니다.

사용자의 응용프로그램은 종료하기 전에 명시적으로 트랜잭션을 종료시켜야 합니다. 트랜잭션을 명시적으로 종료시키지 않으면 프로그램이 성공적으로 종료될 때 프로그램의 보류 트랜잭션 중에 이루어진 모든 변경사항을 DB2이 자동으로 확약합니다. 단, Windows32비트 운영 체제에서는 예외입니다. DB2은 다음 조건 하에서의 변경사항은 구간 복원합니다.

- 로그 전체 조건
- 데이터베이스 관리 프로그램 처리가 종료되도록 하는 기타 시스템 조건

Windows 32비트 운영 체제에서는 트랜잭션을 명시적으로 확약하지 않으면 데이터베이스 관리 프로그램은 항상 트랜잭션을 구간 복원합니다.

프로그램 종료에 대한 세부사항은『프로그램 종료』및 133 페이지의『진단 조절 및 SQLCA 구조』의 내용을 참조하십시오.

ROLLBACK문 사용: 이 명령문은 현재 트랜잭션을 종료하고, 트랜잭션을 시작하기 전 상태로 데이터를 복원합니다.

ROLLBACK문은 호스트 변수 내용에는 영향을 미치지 않습니다.

오류 또는 경고로 인해 입력된 루틴에서 ROLLBACK문을 사용하고 SQL WHENEVER문을 사용하면, ROLLBACK 이전에 WHENEVER SQLERROR CONTINUE 및 WHENEVER SQLWARNING CONTINUE를 지정해야 합니다. 이렇게 하면 ROLLBACK이 오류나 경고로 인해 실패할 경우 프로그램 루프가 발생하는 것을 방지할 수 있습니다.

심각한 오류가 발생하면 ROLLBACK문을 실행할 수 없다는 메시지가 나타납니다. 클라이언트와 서버 응용프로그램 사이의 통신 두절 또는 데이터베이스의 손상과 같은 심각한 오류가 발생할 경우 ROLLBACK문을 실행하지 마십시오. 심각한 오류가 발생한 후 실행할 수 있는 유일한 명령문은 CONNECT문입니다.

프로그램 종료

프로그램을 제대로 종료하려면,

1. (진행중일 경우) COMMIT문 또는 ROLLBACK문을 명시적으로 실행하여 현재 트랜잭션을 종료하십시오.
2. CONNECT RESET문을 사용하여 데이터베이스 서버로의 연결을 해제하십시오.
3. 프로그램이 사용하는 자원을 제거하십시오. 예를 들어, 사용 중인 데이터 구조 또는 임시 저장영역을 비우십시오.

주: 프로그램이 종료될 때 현재 트랜잭션이 여전히 활동중인 경우 DB2는 트랜잭션을 내재적으로 종료합니다. 내재적으로 트랜잭션을 종료할 때 DB2 작동이 플랫폼별로 이루어질 경우, 프로그램이 종료되기 전에 COMMIT문 또는 ROLLBACK문을 실행하여 모든 트랜잭션을 명시적으로 종료해야 합니다. DB2가 내재적으로 트랜잭션을 종료하는 방법에 대한 세부사항은 트랜잭션 내재적 종료의 내용을 참조하십시오.

트랜잭션 내재적 종료

사용자 프로그램이 현재 트랜잭션을 끝내지 않고 종료되는 경우 DB2는 내재적으로 현재 트랜잭션을 종료시킵니다(프로그램의 올바른 종료 방법에 대해서는 23 페이지의 『프로그램 종료』의 내용을 참조하십시오). DB2는 응용프로그램이 종료될 때 COMMIT문이나 ROLLBACK문을 실행하여 현재 트랜잭션을 종료시킵니다. DB2가 COMMIT문 또는 ROLLBACK문 중에서 어느 것을 실행할 것인지는 다음과 같은 인수들에 의해 결정됩니다.

- 응용프로그램의 정상 종료 여부
- DB2 서버가 수행되는 플랫폼
- 응용프로그램의 문맥 API 사용 여부(607 페이지의 『다중 스레드 데이터베이스 액세스』 참조)

대부분의 지원 운영 체제에서

DB2는 종료가 정상적으로 이루어질 경우 트랜잭션을 내재적으로 확약하고, 비정상적으로 이루어질 경우 트랜잭션을 내재적으로 구간 복원합니다. 사용자 프로그램이 비정상 종료라고 여기는 것은 데이터베이스 관리 프로그램이 비정상 종료라고 간주하지 않습니다. 예를 들어, 응용프로그램에 예기치 못한 오류가 발생하여 응용프로그램이 갑자기 종료될 때 `exit(-16)`를 코드화할 수 있습니다. 데이터베

이스 관리 프로그램은 이를 정상 종료라고 간주하고 트랜잭션을 확약합니다. 데이터베이스 관리 프로그램은 분할 위반 또는 예외와 같은 항목들을 비정상 종료로 여깁니다.

Windows 32비트 운영 체제

DB2는 사용자가 명시적으로 COMMIT문을 사용하여 트랜잭션을 명시적으로 확약하지 않는 한, 응용프로그램이 정상적으로 종료되었는지에 관계없이 항상 트랜잭션을 구간 복원합니다.

DB2 Context API 사용시

사용자 응용프로그램은 DB2 API를 사용하여 607 페이지의 『다중 스레드 데이터베이스 액세스』에 설명된 대로 스레드 사이에 응용프로그램 문맥들을 설정 및 전달할 수 있습니다. 사용자 응용프로그램이 이러한 DB2 API를 사용할 경우 DB2는 응용프로그램이 정상적으로 종료되는지에 관계없이 트랜잭션을 내재적으로 구간 복원합니다. COMMIT문을 사용하여 트랜잭션을 명시적으로 확약하지 않는 한 DB2는 트랜잭션을 구간 복원합니다.

응용프로그램 의사코드 프레임워크

프로그램 코딩용 의사코드 프레임워크에서는 DB2 응용프로그램용 일반 프레임워크를 의사코드 형식으로 요약 설명합니다. 물론 사용자 프로그램에 맞도록 이 프레임워크를 조정해야 합니다.

```
Start Program
EXEC SQL BEGIN DECLARE SECTION                                    |
  DECLARE USERID FIXED CHARACTER (8)                              |
  DECLARE PW FIXED CHARACTER (8)                                  |
                                                                  | 응용프로그램
  (other host variable declarations)                              | 설정
                                                                  |
EXEC SQL END DECLARE SECTION                                      |
EXEC SQL INCLUDE SQLCA                                            |
EXEC SQL WHENEVER SQLERROR GOTO ERRCHK                            |

  (program logic)

EXEC SQL CONNECT TO database A USER :userid USING :pw             |
EXEC SQL SELECT ...                                               |
EXEC SQL INSERT ...                                               | 첫번째
  (more SQL statements)                                           | UOW
EXEC SQL COMMIT                                                   |

  (more program logic)

EXEC SQL CONNECT TO database B USER :userid USING :pw             |
EXEC SQL SELECT ...                                               |
EXEC SQL DELETE ...                                               | 두 번째
```

```
  (more SQL statements)                          | UOW
EXEC SQL COMMIT                                  |

  (more program logic)

EXEC SQL CONNECT TO database A                   |
EXEC SQL SELECT ...                              |
EXEC SQL DELETE ...                              | 세 번째
  (more SQL statements)                          | UOW
EXEC SQL COMMIT                                  |

  (more program logic)

EXEC SQL CONNECT RESET                           |
ERRCHK                                           |
                                                 | 응용프로그램
  (check error information in SQLCA)             | 소거
                                                 |
End Program
```

DB2용 응용프로그램 설계

DB2는 다양한 응용프로그램 개발 기능을 제공하여 사용자가 기존의 응용프로그램 기능을 보충 및 확장시킬 수 있도록 합니다. 사용자가 응용프로그램 설계자라면 가장 기본적인 설계, 즉 *응용프로그램 설계에 어떤 DB2 기능을 사용할 것인지*를 결정해야 합니다. 적절한 선택을 위해서는 사용자 응용프로그램에 대한 데이터베이스 설계 및 목표 환경 모두를 고려해야 합니다. 예를 들어, 응용프로그램에 논리를 포함시키는 대신 데이터베이스 설계에 일부 비지니스 규칙을 강화시키도록 선택할 수 있습니다.

사용 기능과 사용 정도는 상당히 유동적입니다. 이 절에서는 사용자 설계에 중요한 영향을 미칠 수 있는 사용가능한 기능들에 대한 개요와, 어떤 항목을 다른 항목 대신 선택하게 되는 이유를 제공합니다. 설명된 기능에 대한 세부사항에 대해서는 참조 내용이 제공됩니다.

다음과 같은 기능들을 고려해야 합니다.

- 다음을 사용한 데이터 액세스:
  - Embedded SQL, Java용 embedded SQLJ(SQLJ) 포함
  - DB2 Call Level Interface(DB2 CLI), Open Database Connectivity(ODBC) 및 Java Database Connectivity(JDBC)
  - Microsoft 스펙
  - Perl DBI

- 조회 제품
- 다음을 사용한 데이터 값 제어:
  - 데이터 유형(내장 또는 사용자 정의)
  - 테이블 점검 제한조건
  - 참조 무결성 제한조건
  - CHECK OPTION을 사용한 뷰
  - 응용프로그램 논리 및 변수 유형
- 다음을 사용한 데이터 값들 간의 관계 제어:
  - 참조 무결성 제한조건
  - 트리거
  - 응용프로그램 논리
- 다음을 사용한 서버에서의 프로그램 실행:
  - 저장 프로시듀어
  - 사용자 정의 함수
  - 트리거

이 목록에서는 트리거와 같은 일부 기능들이 한 번 이상 언급되어 있음을 발견하게 될 것입니다. 이는 하나 이상의 설계 기준을 주소지정하는 기능의 융통성을 반영한 것입니다.

최초의 가장 기본적인 의사결정은 데이터에 대한 응용프로그램 관련 규칙을 강화하기 위해 논리를 데이터베이스로 이동시킬 것인지 여부입니다.

데이터에 집중된 논리를 응용프로그램에서 데이터베이스로 이동시키는 이점은 사용자 응용프로그램이 보다 데이터 독립적이 된다는 것입니다. 데이터를 둘러싸고 있는 논리가 한 장소, 즉 데이터베이스에 집중됩니다. 이는 데이터 또는 데이터 논리를 한 번 변경하면 즉시 **모든** 응용프로그램이 변경된다는 것을 의미합니다.

이는 매우 강력한 장점이긴 하지만 데이터베이스에 입력되는 데이터 논리가 그 데이터의 **모든** 사용자에게 동등하게 영향을 준다는 사실을 명심해야 합니다. 데이터에 적용시키려는 규칙 및 제한조건을 모든 데이터 사용자에게 적용시킬 것인지 응용프로그램 사용자에게만 제한할 것인지를 고려해야 합니다.

사용자의 응용프로그램 요건은 데이터베이스 또는 응용프로그램에서의 규칙을 강화할 것인지에도 영향을 줄 수 있습니다. 예를 들어, 특정 절차로 데이터 입력의 검증 오류를 처리해야 할 경우도 있습니다. 일반적으로 이러한 유형의 데이터 검증은 응용프로그램 코드로 수행해야 합니다.

또한, 응용프로그램이 사용되는 컴퓨팅 환경도 고려해야 합니다. 클라이언트 머신에서 수행되는 논리와, 저장 프로시듀어나 UDF의 사용또는 이 둘을 함께 사용하여 일반적으로 보다 강력한 데이터베이스 서버머신에서 논리를 수행하는 것의 차이점을 고려해야 합니다.

경우에 따라, (응용프로그램별 요건으로 인한) 응용프로그램과 (응용프로그램 외부에서의 기타 대화식 사용으로 인한) 데이터베이스 모두에 강화된 기능을 포함시키는 것이 맞는 답이 되는 경우가 있습니다.

데이터로 액세스

관계형 데이터베이스에서는 SQL을 사용하여 원하는 데이터에 액세스해야 하지만, SQL을 응용프로그램에 통합시키는 방법을 선택할 수도 있습니다. 다음 인터페이스 및 지원 언어로부터 선택할 수도 있습니다.

Embedded SQL
: C/C++, COBOL, FORTRAN, Java (SQLJ), REXX

DB2 CLI 및 ODBC
: C/C++, Java (JDBC)

Microsoft 스펙, ADO, RDO 및 OLE DB 포함
: Visual Basic, Visual C++

Perl DBI
: Perl

Query Products
: Lotus Approach, IBM Query Management Facility

Embedded SQL

Embedded SQL에는 정적 SQL이나 동적 SQL 또는 이 두 유형의 혼합으로 구성될 수 있다는 이점이 있습니다. 응용프로그램이 사용중일 때 SQL문 내용 및 형

식이 동결되면, 응용프로그램에서 embedded 정적 SQL을 사용해 보는 것이 좋습니다. 정적 SQL을 이용하여 응용프로그램을 실행하는 사람은 응용프로그램을 데이터베이스에 바인드한 사용자의 특권을 임시로 물려받습니다. DYNAMICRULES BIND 옵션을 이용하여 응용프로그램을 바인드하지 않는 한 동적 SQL은 응용프로그램을 실행하는 사람의 특권을 사용합니다. 일반적으로 런타임 실행 명령문이 결정되는 Embedded 동적 SQL을 사용해야 합니다. 그러면 보다 다양한 입력을 처리할 수 있는 보다 안전한 응용프로그램이 작성됩니다.

주: Java용 Embedded SQL(SQLJ) 응용프로그램에서는 정적 SQL문만 포함할 수 있습니다. 그러나 JDBC를 사용하여 SQLJ 응용프로그램에서 동적 SQL 호출을 할 수 있습니다.

사용자의 프로그래밍 언어 컴파일러를 사용하기 전에 SQL문을 호스트 언어 명령으로 변환하려면 Embedded SQL 응용프로그램을 사전 처리 컴파일해야 합니다. 또한, 응용프로그램이 수행되기 위해서는 응용프로그램에 있는 SQL을 데이터베이스로 바인드해야 합니다.

Embedded SQL 사용에 대한 세부사항은 75 페이지의『제4장 정적 SQL 프로그램 작성』의 내용을 참조하십시오.

REXX 고려사항: REXX 응용프로그램은 데이터베이스 관리 프로그램 API 및 SQL이 제공하는 대부분의 기능을 사용할 수 있도록 하는 API를 사용합니다. 컴파일된 언어로 작성된 응용프로그램과 달리 REXX 응용프로그램은 사전 처리 컴파일되지 않습니다. 대신, 동적 SQL 핸들러가 모든 SQL문을 처리합니다. 이러한 호출가능 API와 REXX를 결합함으로써 대부분의 데이터베이스 관리 프로그램 기능에 액세스할 수 있습니다. REXX가 Embedded SQL을 사용하여 일부 API를 간접 지원하지 않아도 REXX 응용프로그램 내부로부터 DB2 Command Line Processor를 사용하여 액세스할 수 있습니다.

REXX가 해석 언어이므로, 컴파일된 호스트 언어와 비교해 볼 때 사용자 응용프로그램 프로토타입을 REXX로 개발 및 디버그하기가 보다 쉬울 것입니다. REXX로 코드화된 DB2 응용프로그램이 컴파일된 언어를 사용하는 DB2 응용프로그램의 성능을 제공하지 않지만, 추가 소프트웨어를 사용하거나 사전 처리 컴파일, 컴파일, 링크 등을 하지 않고 DB2 응용프로그램을 작성하는 기능을 제공합니다.

REXX를 사용한 DB2 응용프로그램 코드화 및 구축에 대한 세부사항은 805 페이지의 『제25장 REXX 프로그래밍』의 내용을 보십시오.

DB2 Call Level Interface(DB2 CLI) 및 Open Database Connectivity(ODBC)

DB2 Call Level Interface(DB2 CLI)는 데이터베이스 서버의 DB2 계열에 대한 IBM의 호출가능 SQL 인터페이스입니다. 관계형 데이터베이스 액세스에 대한 C 및 C++ 응용프로그래밍 인터페이스로서, 함수 호출을 사용하여 함수 인수로서 동적 SQL문을 전달합니다. 호출가능 SQL 인터페이스는 데이터베이스 액세스용 API로서, 함수 호출을 사용하여 동적 SQL문을 호출합니다. 이는 Embedded 동적 SQL 대신 사용할 수 있지만, Embedded SQL과는 달리 사전처리 컴파일이나 바인딩은 필요하지 않습니다.

DB2 CLI는 Microsoft™ Open Database Connectivity(ODBC) 스펙 및 X/Open® 스펙에 기초합니다. IBM은 산업 표준을 따르고, 이들 데이터베이스 인터페이스 중 하나에 익숙한 DB2 응용프로그램 프로그래머에 대한 보다 단기간의 학습 곡선을 제공하기 위해 이 스펙을 선택했습니다.

DB2에서의 ODBC 지원에 대한 세부사항은 *CLI Guide and Reference*의 내용을 보십시오.

JDBC

DB2 Java 지원에는 JDBC가 포함됩니다. JDBC는 표준화된 Java 메소드를 통해 응용프로그램에 대한 데이터 액세스를 제공하는 제조업체에 상관없는 동적 SQL 인터페이스입니다. JDBC는 JDBC 프로그램을 사전처리 컴파일하거나 바인드할 필요가 없다는 점에서 DB2 CLI와 유사합니다. 제조업체 중립 표준으로서 JDBC 응용프로그램은 증가된 이식성을 제공합니다.

JDBC를 사용하여 작성된 응용프로그램은 동적 SQL만 사용합니다. JDBC 인터페이스는 추가 처리 오버헤드를 부과합니다.

JDBC에 대한 세부사항은 718 페이지의 『JDBC 프로그래밍』의 내용을 참조하십시오.

Microsoft 스펙

Microsoft Visual Basic™ 또는 Visual C++™로 ActiveX Data Object(ADO)에 일치하는 데이터베이스 응용프로그램을 작성할 수 있습니다. ADO 응용프로그램은 OLE DB Bridge를 사용합니다. Visual Basic으로 Remote Data Object(RDO) 스펙을 따르는 데이터베이스 응용프로그램을 작성할 수 있습니다. 또한 OLE DB 제공자로부터 데이터를 리턴하는 OLE DB 테이블 함수도 정의할 수 있습니다. OLE DB 테이블 함수에 대한 세부사항은 479 페이지의『OLE DB 테이블 함수』의 내용을 참조하십시오.

이 책에는 ADO 및 RDO 스펙을 따르는 응용프로그램 작성에 대한 안내서를 제공하지 않습니다. ADO 및 RDO 스펙을 사용하는 DB2 응용프로그램의 전체 샘플 에 대해서는 다음 디렉토리를 참조하십시오.

- Visual Basic으로 작성된 샘플은 `sqllib\samples\VB`에서 자세한 내용을 참조하십시오.
- Visual C++로 작성된 샘플은 `sqllib\samples\VC`에서 자세한 내용을 참조하십시오.
- RDO 스펙을 사용하는 샘플은 `sqllib\samples\RDO`에서 자세한 내용을 참조하십시오.
- Microsoft Transaction Server™를 사용하는 샘플은 `sqllib\samples\MTS`에서 자세한 내용을 참조하십시오.

Perl DBI

DB2는 DBD::DB2 드라이버를 통해 데이터 액세스에 대한 Perl Database Interface(DBI) 스펙을 지원합니다. DB2 Database를 액세스하는 Perl DBI와 함께 응용프로그램을 작성하는 정보에 대해서는 755 페이지의『제22장 Perl 프로그래밍』에서 자세한 내용을 참조하십시오. `http://www.ibm.com/software/data/db2/perl/`에 있는 DB2 Universal Database Perl DBI 웹사이트에는 최신 DBD::DB2 드라이버 및 정보가 포함됩니다.

Query Products

IBM Query Management Facility(QMF) 및 Lotus Notes가 포함된 조회 제품은 조회 개발 및 보고를 지원합니다. 제품은 SQL문 개발 방법과 소개될 수 있는

논리 등급에 따라 달라집니다. 사용자 필요에 따라 이 접근 방식은 데이터 액세스를 위한 요건을 충족시킬 수 있습니다. 이 책은 조회 제품에 대한 자세한 정보를 제공하지 않습니다.

데이터 값 제어

응용프로그램 논리의 전형적인 영역은 데이터베이스에 허용된 값을 제어하여 데이터를 검증 및 보호하는 것입니다. 응용프로그램에는 검증을 위해 입력될 때 데이터 값을 구체적으로 점검하는 논리가 있습니다(예를 들어, 부서 번호가 유효한 번호인지와 기존 부서를 참조하는지를 점검합니다). 데이터베이스 내부가 아닌, DB2의 같은 기능을 제공하는 여러 개의 다른 방식이 있습니다.

데이터 유형

데이터베이스는 테이블의 컬럼에 모든 데이터 요소들을 저장하고, 한 가지 데이터 유형으로 각 컬럼을 정의합니다. 이 데이터 유형은 컬럼에 대한 값의 유형에 일부 제한을 둡니다. 예를 들어, 정수는 고정된 범위 내의 번호여야 합니다. SQL문에서의 컬럼 사용은 특정 작동을 따라야 합니다. 예를 들어, 데이터베이스는 정수를 문자열과 비교하지 않습니다. DB2에는 정의된 특성 및 작동으로 내장된 데이터 유형 세트가 포함됩니다. DB2는 *사용자 정의 구별 유형*이라고 하는 사용자 자신의 데이터 유형 정의도 지원합니다. 이 유형은 내장된 유형에 기초하지만 내장된 유형의 모든 작동을 자동으로 지원하지 않습니다. 또한, 2진 대형 오브젝트(BLOB)와 같은 데이터 유형을 사용하여, 데이터 구조와 같은 관련있는 값 세트로 구성될 수 있는 데이터를 저장할 수도 있습니다.

데이터 유형에 대한 추가 정보는 *SQL 참조서*의 내용을 참조하십시오.

고유 제한조건

고유 제한조건은 테이블 내에서 하나 이상의 컬럼에 이중 값이 발생하는 것을 방지합니다. 고유 키 및 기본 키가 지원되는 고유 제한조건입니다. 예를 들어, 두 부서에 동일한 부서 번호가 지정되지 않도록 하기 위해서는 DEPARTMENT 테이블의 DEPTNO 컬럼에 대한 고유 제한조건을 정의할 수 있습니다.

테이블에 있는 데이터를 사용하는 모든 응용프로그램의 고유 규칙을 강화시키려면 고유 제한조건을 사용하십시오. 고유 제한조건에 대한 추가 정보는 *SQL 참조서*의 내용을 참조하십시오.

테이블 점검 제한조건

테이블 점검 제한조건을 사용하여 데이터 유형을 넘어서, 테이블의 컬럼에 허용되는 값에 대한 제한사항을 정의할 수 있습니다. 테이블 점검 제한조건은 같은 테이블의 같은 행에 있는 다른 값들에 대한 범위 점검 형식을 취합니다.

규칙이 데이터를 사용하는 모든 응용프로그램에 적용되는 경우, 테이블 점검 제한조건을 사용하여 테이블에 허용되는 데이터에 대한 제한사항을 강화하십시오 테이블 점검 제한조건은 제한사항을 일반적으로 적용가능하며 유지보수가 보다 쉬워지도록 합니다.

테이블 점검 제한조건에 대한 추가 정보는 *SQL 참조서*의 내용을 참조하십시오.

참조 무결성 제한조건

데이터를 사용하는 모든 응용프로그램에 대한 값 기초 관계를 유지보수해야 할 경우 참조 무결성(RI) 제한조건을 사용하십시오. 예를 들어, RI 제한조건을 사용하여 EMPLOYEE 테이블의 DEPTNO 컬럼 값이 DEPARTMENT 테이블의 값과 일치하는지 확인하십시오. 이 제한조건은 DEPARTMENT 정보를 누락시킬 수 있는 삽입, 갱신 또는 삭제 조작을 예방합니다. 데이터베이스에서 사용자 규칙을 중앙화함으로써 RI 제한조건은 일반적으로 규칙을 적용가능하게 하고 유지보수가 보다 쉽도록 합니다.

RI 제한조건의 심화 사용법에 대해서는 34 페이지의 『데이터 관계 제어』의 내용을 참조하십시오.

참조 무결성에 대한 추가 정보는 *SQL 참조서*의 내용을 참조하십시오.

점검 옵션 보기

응용프로그램이 테이블 점검 제한조건으로서 원하는 규칙을 정의할 수 없거나 데이터의 모든 사용에 규칙이 적용되지 않는 경우, 응용프로그램 논리에 규칙을 두는 또다른 대안이 있습니다. WHERE절 및 WITH CHECK OPTION절의 일부로서 지정된 데이터에 대한 조건으로 테이블 뷰를 작성해 볼 수 있습니다. 이 뷰 정의는 사용자 응용프로그램에 유용한 세트로 데이터 검색을 제한합니다. 또한 뷰를 갱신할 수 있는 경우, WITH CHECK OPTION절은 갱신, 삽입, 삭제를 응용프로그램에 적용가능한 행으로 제한합니다.

WITH CHECK OPTION에 대한 추가 정보는 *SQL 참조서*의 내용을 참조하십시오.

응용프로그램 논리 및 프로그램 변수 유형

프로그래밍 언어로 응용프로그램 논리를 작성할 때 위에서 설명한 데이터에 대해 같은 제한사항 중 일부를 제공하기 위해 변수를 선언하기도 합니다. 또한, 데이터베이스 대신 응용프로그램에서 규칙 강화 코드를 작성하도록 선택할 수 있습니다. 다음과 같은 경우, 응용프로그램 서버(AS)에 논리를 두십시오.

- 33 페이지의 『점검 옵션 보기』에 표시된 뷰의 경우를 제외하고는 일반적으로 규칙이 적용되지 않는 경우.
- 데이터베이스의 데이터의 정의에 대한 제어 권한이 없는 경우.
- 응용프로그램 논리로 보다 효율적으로 규칙을 처리할 수 있는 경우.

예를 들어, 입력 절차로 입력 데이터의 오류 처리가 필요하지만 데이터베이스내의 조작 절차로 보증할 수는 없습니다.

데이터 관계 제어

응용프로그램 논리에 또다른 주요 촛점 영역은 사용자 시스템의 서로 다른 논리 엔터티 사이의 관계 관리 영역에 있습니다. 예를 들어, 새로운 부서를 추가하려면 새로운 계정 코드를 작성해야 합니다. DB2는 사용자 데이터베이스의 다른 오브젝트들 사이의 관계를 관리하는 두 가지 방법, 즉 참조 무결성 제한조건과 트리거를 제공합니다.

참조 무결성 제한조건

데이터 관계 제어의 관점으로부터 고려되는 참조 무결성(RI) 제한조건을 통해 한 개 이상의 테이블에 있는 데이터 사이의 관계를 제어할 수 있습니다. CREATE TABLE문 또는 ALTER TABLE문을 사용하여 DELETE문과 UPDATE문과 같은 관련된 1차 키에 영향을 주는 작업의 작동을 정의할 수 있습니다.

RI 제한조건은 하나 이상의 테이블을 통해 데이터에 대한 사용자 규칙을 강화시킵니다. 규칙이 이 데이터를 사용하는 모든 응용프로그램에 적용될 경우 RI 제한조건은 데이터베이스에 규칙들을 중앙화시킵니다. 이를 통해 일반적으로 규칙들은 적용가능해지고 유지보수가 보다 쉬워집니다.

참조 무결성에 대한 추가 정보는 *SQL 참조서*의 내용을 참조하십시오.

트리거

갱신 전후에 트리거를 사용하여 응용프로그램에서도 수행될 수 있는 논리를 지원할 수 있습니다. 트리거에 의해 지원되는 조작 또는 규칙이 데이터를 사용하는 모든 응용프로그램에 적용되는 경우, 트리거는 데이터베이스에 규칙 또는 조작을 중앙화함으로써, 일반적으로 적용가능하고 유지보수가 보다 쉽게 만듭니다.

트리거에 대한 추가 정보는 537 페이지의『제16장 활동중인 DBMS에서 트리거 사용』및 *SQL 참조서*의 내용을 참조하십시오.

갱신 전 트리거 사용: 갱신 또는 삽입 전에 수행되는 트리거를 사용하여, 갱신되거나 삽입 중인 값들은 실제로 데이터베이스가 수정되기 전에 수정될 수 있습니다. 이는 응용프로그램(데이터의 사용자 뷰)으로부터의 입력을 원하는 내부 데이터베이스 형식으로 변환시키는 데 사용할 수 있습니다. 이 *사전 트리거*는 다른 비-데이터베이스 조작이 사용자 정의 함수를 통해서 작동화되도록 하는 데에도 사용할 수 있습니다.

갱신 후 트리거 사용: 갱신, 삽입 또는 삭제 후 수행되는 트리거를 여러 가지 방식으로 사용할 수 있습니다.

- 트리거는 동일한 테이블이나 기타 테이블의 데이터를 갱신, 삽입 또는 삭제할 수 있습니다. 이는 데이터 사이의 관계를 유지보수하거나 감사 추적 정보를 유지하는 데 유용합니다.
- 트리거는 나머지 테이블이나 기타 테이블에 있는 데이터 값과 데이터를 비교 점검합니다. 이 테이블 또는 기타 테이블의 다른 행으로부터의 데이터에 대한 참조로 인해 RI 제한조건 또는 점검 제한조건을 사용할 수 없을 때 유용합니다.
- 트리거는 사용자 정의 함수를 사용하여 비-데이터베이스 조작을 활성화할 수 있습니다. 예를 들어, 데이터베이스 외부에서 경고를 발행하거나 정보를 갱신할 경우 유용합니다.

응용프로그램 논리

데이터베이스 대신 응용프로그램 내에서의 관련 조작을 수행하거나 규칙을 강화하는 코드를 작성하도록 선택할 수 있습니다. 일반적으로 규칙을 데이터베이스에 적용시킬 수 없는 경우에 이를 수행해야 합니다. 데이터베이스에 있는 데이터의 정

의에 대한 제어가 없거나 응용프로그램 논리가 규칙이나 조작을 보다 효율적으로 처리할 수 있다고 생각할 경우, 응용프로그램에 논리를 두도록 선택할 수도 있습니다.

서버에서의 응용프로그램 논리

DB2가 추가 기능을 제공하는 응용프로그램 설계의 최종 단계는 데이터베이스 서버에서의 응용프로그램 논리 일부를 수행하는 것입니다. 일반적으로 사용자는 이 설계를 선택하여 성능을 향상시킬 수 있지만, 공통 함수를 지원하려면 서버에서 응용프로그램 논리를 수행해야 합니다.

저장 프로시듀어

저장 프로시듀어는 클라이언트 응용프로그램 논리로부터 호출되지만 데이터베이스 서버에서 수행되는 응용프로그램에 대한 루틴입니다. 저장 프로시듀어를 사용하는 가장 일반적인 이유는, 소량의 결과 데이터만 산출하는 데이터베이스 집중 처리를 위한 것입니다. 이를 통해 저장 프로시듀어 실행 중에 네트워크를 통해 대용량의 통신을 저장할 수 있습니다. 여러 응용프로그램에 공통인 조작 세트에 대해 저장 프로시듀어를 사용할 수도 있습니다. 이와 같이 모든 응용프로그램은 동일한 논리를 사용하여 조작을 수행합니다.

저장 프로시듀어에 대한 추가 정보는 207 페이지의 『제7장 저장 프로시듀어』의 내용을 참조하십시오.

사용자 정의 함수

다음을 리턴할 SQL문 내에서의 조작 수행에 사용할 사용자 정의 함수(UDF)를 작성할 수 있습니다.

- 단일 스칼라 값(스칼라 함수)
- 비-DB2 데이터 소스의 테이블. 예를 들면 ASCII 파일 또는 웹 페이지(테이블 함수)

UDF에는 SQL문이 포함될 수 없습니다. UDF는 데이터 값 변환, 하나 이상의 데이터 값에 대한 계산 수행 또는 값의 일부 추출(예를 들어, 대형 오브젝트(LOB)의 일부 추출)과 같은 타스크에 유용합니다.

사용자 정의 함수에 대한 추가 정보는 433 페이지의 『제15장 사용자 정의 함수 (UDF)』의 내용을 참조하십시오.

트리거

35 페이지의 『트리거』에서 트리거는 사용자 정의 함수를 호출하는 데 사용할 수 있습니다. 특정 명령문이 발생하거나 데이터 값이 변경될 때 특정 비-SQL 조작이 항상 수행되기를 원할 때 유용합니다. 예에는 특정 상황에서의 전자 우편 메시지 실행이나 경고 유형 정보를 파일에 기록하는 것과 같은 조작이 포함됩니다.

트리거에 대한 추가 정보는 537 페이지의 『제16장 활동중인 DBMS에서 트리거 사용』의 내용을 참조하십시오.

Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가

Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가는 Visual Studio IDE의 Visual C++ 구성요소에 플러그인하는 관리 도구와 마법사의 컬렉션입니다. 도구와 마법사는 embedded SQL을 사용하여 DB2용 응용프로그램을 개발할 때 이루어지는 다양한 작업을 자동화하고 단순화시킵니다.

Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가를 사용하여 개발, 패키지 및 전개:

- Windows 32-비트 운영 체제의 DB2 Universal Database에 대해 C/C++로 작성된 저장 프로시듀어
- DB2 Universal Database 서버를 액세스하는 Windows 32-비트 C/C++ embedded SQL 클라이언트 응용프로그램
- C/C++ 함수 호출 랩퍼를 사용하여 저장 프로시듀어를 호출하는 Windows 32-비트 C/C++ 클라이언트 응용프로그램

Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가를 통해 실제 DB2 응용프로그램 그축 및 전개 대신 그 설계 및 논리에 촛점을 맞출 수 있습니다.

Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가로 수행된 일부 타스크에는 다음이 포함됩니다.

- 새 embedded SQL 모듈 작성
- SQL 지원을 사용하여 embedded SQL에 SQL문 삽입

- 가져온 저장 프로시듀어 추가
- 내보낸 저장 프로시듀어 작성
- DB2 프로젝트 패키지화
- Visual C++에서의 DB2 프로젝트 전개

Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가는 도구막대의 형식으로 제공됩니다. 도구막대 버튼에는 다음이 포함됩니다.

DB2 프로젝트 특성
: 프로젝트 특성(데이터베이스 개발 및 코드 생성 옵션)을 관리합니다.

새 DB2 오브젝트
: 새 embedded SQL 모듈, 가져온 저장 프로시듀어 또는 내보낸 저장 프로시듀어를 추가합니다.

DB2 Embedded SQL 모듈
: Embedded SQL 모듈의 목록과 사전 처리 컴파일러 옵션을 관리합니다.

DB2 저장 프로시듀어 가져오기
: 가져온 저장 프로시듀어 목록을 관리합니다.

DB2 저장 프로시듀어 내보내기
: 내보낸 저장 프로시듀어 목록을 관리합니다.

DB2 프로젝트 패키지
: DB2 외부 프로젝트 파일을 패키지화합니다.

DB2 프로젝트 전개
: 패키지화된 DB2 외부 프로젝트 파일을 전개합니다.

Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가에는 표준 C++ 도구 사용자 정의 옵션을 사용하여 볼 수 있는 다음과 같은 숨겨진 3개의 버튼이 있습니다.

새 DB2 Embedded SQL 모듈
: 새 C/C++ embedded SQL 모듈을 추가합니다.

새 DB2 저장 프로시듀어 가져오기
: 새 데이터베이스 저장 프로시듀어를 가져옵니다.

새 DB2 저장 프로시듀어 내보내기
: 새 데이터베이스 저장 프로시듀어를 내보냅니다.

Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가는 자동으로 다음과 같은 코드 요소를 생성합니다.

- 선택적 샘플 SQL문이 있는 Skeletal embedded SQL 모듈 파일
- 표준 데이터베이스 연결 및 embedded SQL 함수 연결해제
- 가져온 저장 프로시듀어 호출 랩퍼 함수
- 내보낸 저장 프로시듀어 함수 템플리트
- 내보낸 저장 프로시듀어 데이터 정의 언어 (DDL) 파일

Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가에 연관된 용어:

IDE 프로젝트
: 표준 Visual C++ 프로젝트

DB2 프로젝트
: IDE 프로젝트에 삽입되는 DB2 프로젝트 오브젝트 컬렉션. DB2 프로젝트 오브젝트를 Visual C++ 프로젝트에 삽입할 수 있습니다. DB2 프로젝트를 통해 embedded SQL 모듈, 가져온 저장 프로시듀어 그리고 내보낸 저장 프로시듀어와 같은 다양한 DB2 오브젝트를 관리할 수 있습니다. 이런 오브젝트와 그 특성을 추가, 삭제 그리고 수정할 수 있습니다.

모듈
: SQL문이 들어 있을 수도 있는 C/C++ 소스 코드 파일.

개발 데이터베이스
: Embedded SQL 모듈을 컴파일할 때 사용되는 데이터베이스. 개발 데이터베이스는 중요한 데이터베이스 저장 프로시듀어 정의 목록을 찾아볼 때도 사용됩니다.

embedded SQL 모듈
: Embedded 정적 또는 동적 SQL을 포함하는 C/C++ 소스 코드.

저장 프로시듀어 가져오기
: 이미 데이터베이스에 정의되어 있으며 프로젝트가 호출하는 저장 프로시듀어.

저장 프로시듀어 내보내기
: 프로젝트에서 구축하고 정의되는 데이터베이스 저장 프로시듀어.

Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가 활성화

Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가를 활성화시키려면 다음 단계를 수행하십시오.

단계 1. 아직 추가를 등록하지 않았으면 명령행에 다음을 입력하여

```
db2vccmd register
```

등록하십시오.

단계 2. **도구 —> 사용자 정의**를 선택하십시오. 사용자 정의 노트북이 열립니다.

단계 3. **추가 및 매크로 파일** 탭을 선택하십시오. 추가 및 매크로 파일 페이지가 열립니다.

단계 4. **IBM DB2 프로젝트 추가** 선택란을 선택하십시오.

단계 5. **확인**을 클릭하십시오. 부동 도구막대가 작성됩니다.

주: 도구막대가 실수로 닫히면 추가를 비활성화킨 다음 다시 활성화시키거나 Microsoft Visual C++ 표준 사용자 정의 옵션을 사용하여 도구막대를 다시 표시하십시오.

Microsoft Visual C++에 대한 IBM DB2 Universal Database 도구 추가 활성화

DB2 도구 추가는 Visual C++ 통합 개발 환경에서 일부 DB2 관리 및 개발 도구를 시작하는 도구막대입니다.

Microsoft Visual C++에 대한 IBM DB2 Universal Database 도구 추가를 활성화시키려면 다음 단계를 수행하십시오.

단계 1. 아직 추가를 등록하지 않았으면 명령행에 다음을 입력하여

```
db2vccmd register
```

등록하십시오.

단계 2. **도구 —> 사용자 정의**를 선택하십시오. 사용자 정의 노트북이 열립니다.

단계 3. 추가 및 매크로 파일 탭을 선택하십시오.

단계 4. **IBM DB2 도구 추가** 선택란을 선택하십시오.

단계 5. **확인**을 클릭하십시오. 부동 도구막대가 작성됩니다.

주: 도구막대가 실수로 닫히면 추가를 비활성화시킨 다음 다시 활성화시키거나 Visual C++ 표준 사용자 정의 옵션을 사용하여 도구막대를 다시 표시하십시오.

Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가에 대한 자세한 내용은 다음을 참조하십시오.

- Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가의 온라인 도움말.
- http://www.ibm.com/software/data/db2/udb/ide/index.html.

지원되는 SQL문

SQL 언어는 응용프로그램 내부로부터 데이터 정의, 검색, 갱신 및 제어 조작을 제공합니다. 829 페이지의 표37에는 DB2 제품이 지원하는 SQL문이 표시되어 있고, 이 명령문이 CLP를 통해서 또는 DB2 CLI를 통해서 동적으로 지원되는지를 나타냅니다. 829 페이지의 표37을 빠른 참조서로 사용할 수 있습니다. 모든 명령문 및 구문에 대한 전체 논의는 *SQL 참조서*의 내용을 참조하십시오.

권한 부여 고려사항

*권한 부여*를 통해 사용자 또는 그룹은 데이터베이스에 연결, 테이블 작성 또는 시스템 관리와 같은 일반적인 타스크를 수행할 수 있습니다. *특권*은 지정된 방식으로 특정 데이터베이스 오브젝트 하나에 액세스할 수 있는 권한을 사용자 또는 그룹에게 제공합니다. DB2는 특권 세트를 사용하여 그 안에 저장하는 정보에 대한 보호를 제공합니다. 다른 특권에 대해서는 *관리 안내서: 계획*에서 자세한 내용을 참조하십시오.

대부분의 SQL문에는 명령문이 이용하는 데이터베이스 오브젝트에 대한 일부 특권 유형이 필요합니다. 일반적으로 대부분의 API 호출의 경우, 호출이 이용하는 데이터베이스 오브젝트에 대한 특권이 필요하지 않지만, 대부분의 API에서는 사용자가 이들 오브젝트를 호출하기 위해서는 필요한 권한을 소유해야 합니다. DB2 API

를 이용하여 사용자의 응용프로그램으로부터 DB2 관리 기능을 수행할 수 있습니다. 예를 들어, 바인드 파일에 대한 필요 없이 데이터베이스에 저장된 패키지를 재작성하기 위해 `sqlarbnd`(또는 REBIND) API를 사용할 수 있습니다. 각 DB2 API에 대한 세부사항은 *Administrative API Reference*의 내용을 참조하십시오.

각 SQL문 실행에 필요한 특권에 대해서는 *SQL 참조서*의 내용을 참조하십시오. 각 API 호출 발행에 필요한 특권 및 권한에 대해서는 *Administrative API Reference*의 내용을 참조하십시오.

응용프로그램을 설계할 때 사용자가 응용프로그램을 수행할 때 필요한 특권이 무엇일지 고려하십시오. 사용자에게 필요한 특권은 다음 사항에 따라 결정됩니다.

- 응용프로그램이 JDBC 및 DB2 CLI와 같은 동적 SQL을 사용하는지 또는 정적 SQL을 사용하는지 여부
- 응용프로그램이 사용하는 API

동적 SQL

DYNAMICRULES RUN(기본값)으로 바인드된 패키지에서 동적 SQL을 사용하려면, 동적 SQL 응용프로그램을 수행하는 사람은 수행되는 각 SQL 요청을 실행하는 데 필요한 특권 뿐만 아니라 해당 패키지에 대한 EXECUTE 특권이 있어야 합니다. 특권은 사용자의 권한 부여 ID, 사용자가 그 구성원인 그룹 또는 PUBLIC에 권한 부여될 수 있습니다.

DYNAMICRULES BIND 옵션으로 응용프로그램을 바인드할 경우 DB2는 사용자의 권한 부여 ID를 응용프로그램 패키지와 연결시킵니다. 이를 통해 응용프로그램을 사용하는 어떤 사용자라도 권한 부여 ID와 연결된 특권을 계승받을 수 있습니다.

(내포된 동적 SQL 응용프로그램의 경우) 응용프로그램을 바인드하는 사람은 프로그램에 정적 SQL이 포함되어 있지 않은 경우 데이터베이스에 BINDADD 권한만 필요합니다. 다시, 이 특권은 사용자의 권한 부여 ID, 사용자가 구성원인 그룹 또는 PUBLIC에 권한 부여될 수 있습니다.

동적 SQL 패키지를 DYNAMICRULES BIND 옵션과 바인드할 때 응용프로그램을 수행하는 사용자에게는 그 패키지에 대해 EXECUTE 특권만 필요합니다. 동

적 SQL 응용프로그램과 DYNAMICRULES BIND 옵션을 바인드하려면, 응용프로그램에서 모든 동적 및 정적 SQL문을 수행하는 데 필요한 특권이 있어야 합니다. 사용자에게 SYSADM 또는 DBADM 권한이 있고 패키지를 DYNAMICRULES BIND와 바인드할 경우, OWNER BIND 옵션을 사용하여 다른 권한 부여 ID를 지정하십시오. OWNER BIND는 패키지가 자동으로 동적 SQL문에 대한 SYSADM 또는 DBADM 특권을 계승하지 못하도록 합니다. DYNAMICRULES BIND 및 OWNER BIND에 대한 세부사항은 *Command Reference*의 BIND 명령을 참조하십시오.

정적 SQL

정적 SQL을 사용하려면 응용프로그램을 수행하는 사용자는 그 패키지에 대해 EXECUTE 특권만 필요합니다. 패키지를 구성하는 각 명령문에 대해 특권이 필요하지 않습니다. EXECUTE 특권은 사용자의 권한 부여 ID, 사용자가 그 구성원인 그룹 또는 PUBLIC에 권한 부여될 수 있습니다.

그러나 응용프로그램을 바인딩하는 경우, BINDADD 권한과 응용프로그램의 모든 명령문을 수행하는 데 필요한 특권이 있어야 합니다. VALIDATE RUN이 BIND 시간에 지정되면 이 패키지에서 정적 SQL의 모든 권한 부여가 실패해도 BIND는 실패하지 않으며 이런 명령문은 런타임에 다시 확인됩니다. 응용프로그램을 바인드하는 사람에게는 항상 BINDADD 권한이 있어야 합니다. 명령문을 실행하는 데 필요한 특권은 사용자의 권한 부여 또는 PUBLIC에 권한 부여되어야 합니다. 정적 SQL문을 바인딩할 때 그룹 특권은 사용되지 않습니다. 동적 SQL에서와 마찬가지로, BINDADD 특권은 사용자 권한 부여 ID, 사용자가 그 구성원인 그룹 또는 PUBLIC에 부여될 수 있습니다.

이러한 정적 SQL의 특성은 사용자에게 DB2에 있는 정보로의 액세스에 대한 정확한 제어를 제공합니다. 가능한 적용 방법에 대해서는 이 절 끝에 오는 예를 참조하십시오.

API 사용

DB2가 제공하는 대부분의 API에는 특권 사용이 필요하지만, 호출하기 위해서는 일종의 권한이 필요합니다. 특권이 필요한 API의 경우 특권은 응용프로그램을 수행하는 사용자에게 권한 부여되어야 합니다. 특권은 사용자의 권한 부여 ID, 사용

자가 그 구성원인 그룹 또는 PUBLIC에 권한 부여될 수 있습니다. 각 API 호출 발행에 필요한 특권 및 권한에 대해서는 *Administrative API Reference*의 내용을 참조하십시오.

예

STAFF 테이블에 대해 조회를 수행해야 하는 두 사용자, 즉 PAYROLL 및 BUDGET을 고려하십시오. PAYROLL은 회사 직원의 급여를 담당하고 있으므로, 급여 명세표를 발행할 때 다양한 SELECT문을 발행해야 합니다. PAYROLL은 각 사원의 급여에 액세스할 수 있어야 합니다. BUDGET은 지급할 급여액을 결정합니다. 그러나 BUDGET은 특정 사원의 급여는 볼 수 없습니다.

PAYROLL이 여러 가지 많은 SELECT문을 발행하므로 PAYROLL용으로 설계하는 응용프로그램은 동적 SQL을 잘 이용할 수 있어야 합니다. PAYROLL에게는 STAFF 테이블에 대한 SELECT 특권이 필요합니다. PAYROLL은 테이블에 대한 전체 액세스가 필요하므로 이것은 문제가 되지 않습니다.

한편, BUDGET은 각 사원의 급여에 액세스할 수 없습니다. 이는 STAFF 테이블에 대한 SELECT 특권을 BUDGET에 권한 부여할 수 없음을 의미합니다. BUDGET은 STAFF 테이블에 있는 전체 급여의 합계에 액세스하므로 정적 SQL 응용프로그램을 구축하여 SELECT SUM(SALARY) FROM STAFF를 실행하고, 응용프로그램을 바인드하며, 응용프로그램의 패키지에 대한 EXECUTE 특권을 BUDGET에 권한 부여할 수 있습니다. 이렇게 하면 BUDGET이 볼 수 없는 정보를 노출시키지 않고 BUDGET이 필요한 정보를 얻을 수 있습니다.

Embedded SQL 또는 DB2 CLI 프로그램에 사용되는 데이터베이스 관리 프로그램 API

사용자 응용프로그램은 API를 사용하여 SQL문을 사용하여 제공되지 않는 데이터베이스 관리 프로그램 기능에 액세스할 수 있습니다. 데이터베이스 관리 프로그램에서 사용가능한 API에 대한 전체 세부사항 및 호출 방법에 대해서는 *Administrative API Reference*에 있는 예를 참조하십시오.

DB2 API를 사용하여 다음을 수행할 수 있습니다.

- 데이터베이스 및 노드의 카탈로그 및 카탈로그 해제, 데이터베이스 및 노드 디렉토리의 검색이 포함된 데이터베이스 관리 프로그램 환경을 조작합니다. 또한, API를 사용하여 데이터베이스를 작성, 삭제 및 이주시킬 수도 있습니다.
- 데이터 가져오기 및 내보내기 데이터, 데이터베이스의 관리, 백업 및 복원 기능을 제공합니다.
- 데이터베이스 관리 프로그램 구성 파일 및 데이터베이스 구성 파일을 조작합니다.
- 클라이언트/서버 환경별 조작을 제공합니다.
- 사전처리 컴파일된 SQL문에 대한 런타임 인터페이스를 제공합니다. 이 API가 항상 프로그래머에 의해 직접 호출되는 것은 아닙니다. 대신, 처리 후 사전 처리 컴파일러에 의해 수정된 소스 파일로 삽입됩니다.

데이터베이스 관리 프로그램에는 고유의 사전 처리 컴파일러 및 응용프로그램 개발에 유용한 기타 API를 작성하려는 언어 벤더에 대한 API가 포함됩니다.

데이터베이스 관리 프로그램에서 사용가능한 API에 대한 세부사항 및 호출 방법에 대해서는 *Administrative API Reference*에 있는 예를 참조하십시오.

테스트 환경 설정

다음 절에 설명된 대부분의 타스크를 수행하려면 테스트 환경을 설정해야 합니다. 예를 들어, 응용프로그램의 SQL 코드를 테스트하려면 데이터베이스가 필요합니다.

테스트 환경에는 다음 사항이 포함되어야 합니다.

- **테스트 데이터베이스.** 응용프로그램이 테이블 및 뷰로부터 데이터를 갱신, 삽입 또는 삭제하려면 테스트 데이터를 사용하여 실행을 검증하십시오. 테이블 및 뷰로부터 데이터를 검색만 하려면 테스트할 때 생산 레벨의 데이터를 사용해 보십시오.
- **입력 데이터를 테스트하십시오.** 응용프로그램 테스트에 사용되는 입력 데이터는 가능한 모든 입력 조건을 나타내는 유효한 데이터여야 합니다. 응용프로그램이 입력 데이터가 유효한지 검증하려면 유효 데이터 및 무효 데이터 모두를 포함시켜 유효한 데이터가 처리되는지와 유효하지 않은 데이터가 플래그되는지 확인하십시오.

테스트 데이터베이스 작성

테스트 데이터베이스를 작성해야 할 경우 CREATE DATABASE API를 호출하는 작은 서버 응용프로그램을 작성하거나 명령행 처리기을 사용하십시오. 명령행 처리기에 대한 정보는 *Command Reference*의 내용을, CREATE DATABASE API에 대해서는 *Administrative API Reference*의 내용을 참조하십시오.

테스트 테이블 작성

필요한 테스트 테이블 및 뷰를 설계하려면 우선 응용프로그램의 데이터 필요를 분석하십시오. 테이블을 작성하려면 스키마에 대한 CREATETAB 권한 및 CREATEIN 특권이 필요합니다. 대체 권한에 대해서는 *SQL 참조서*에 있는 CREATE TABLE문에 대한 정보를 참조하십시오.

응용프로그램이 액세스하는 데이터를 나열하고 각 데이터 항목이 액세스되는 방식을 설명합니다. 예를 들어, 개발 중인 응용프로그램이 TEST.TEMPL, TEST.TDEPT 및 TEST.TPROJ 테이블에 액세스한다고 가정하십시오. 표1에 표시된 대로 액세스 유형을 기록할 수 있습니다.

표 1. 응용프로그램 데이터 설명

테이블 또는 뷰 이름	행 삽입	행 삭제	컬럼 이름	데이터 유형	액세스 갱신
TEST.TEMPL	아니오	아니오	EMPNO LASTNAME WORKDEPT PHONENO JOBCODE	CHAR(6) VARCHAR(15) CHAR(3) CHAR(4) DECIMAL(3)	예 예 예
TEST.TDEPT	아니오	아니오	DEPTNO MGRNO	CHAR(3) CHAR(6)	
TEST.TPROJ	예	예	PROJNO DEPTNO RESPEMP PRSTAFF PRSTDATE PRENDATE	CHAR(6) CHAR(3) CHAR(6) DECIMAL(5,2) DECIMAL(6) DECIMAL(6)	예 예 예 예 예

응용프로그램 데이터 액세스에 대한 설명이 완료되면 응용프로그램 테스트에 필요한 테스트 테이블 및 뷰를 구성하십시오.

- 테이블이나 뷰에 있는 데이터를 응용프로그램이 수정할 때 테스트 테이블을 작성하십시오. CREATE TABLE SQL문을 사용하여 다음과 같은 테스트 테이블을 작성하십시오.
  - TEMPL
  - TPROJ
- 응용프로그램이 산출 데이터베이스의 데이터를 수정하지 않을 때 테스트 뷰를 작성하십시오.

  이 예에서는 CREATE VIEW SQL문을 사용하여 TDEPT 테이블의 테스트 뷰를 작성하십시오.

데이터베이스 스키마가 응용프로그램을 따라 개발중인 경우 테스트 테이블의 정의는 개발 과정 중에 반복적으로 정제될 수 있습니다. 일반적으로 1차 응용프로그램은 테이블을 작성하거나 액세스할 수 없습니다. 데이터베이스 관리 프로그램은 존재하지 않는 테이블 및 뷰를 참조하는 명령문을 바인드할 수 없습니다. 테이블 작성 및 변경 프로세스 시간이 덜 걸리도록 하려면 별도의 응용프로그램을 개발하여 테이블을 작성해 보십시오. 물론 명령행 처리기(CLP)를 사용하여 테스트 테이블을 대화식으로 작성할 수 있습니다.

테스트 데이터 생성

데이터를 테이블에 삽입하려면 다음 방법 중 하나를 사용하십시오.

- INSERT...VALUES(SQL문)는 명령이 발행될 때마다 하나 이상의 행을 테이블에 둡니다.
- INSERT...SELECT는 (SELECT절에 기초하여) 기존 테이블로부터 데이터를 얻어 이를 INSERT문으로 식별되는 테이블에 둡니다.
- IMPORT 또는 LOAD 유틸리티는 정의된 소스로부터 새로운 또는 기존의 데이터를 대량 삽입합니다.
- RESTORE 유틸리티는 원래 데이터베이스의 BACKUP 사본을 사용하여 기존 데이터베이스 내용을 동일한 테스트 데이터베이스로 복제하는 데 사용할 수 있습니다.

INSERT문에 대해서는 *SQL 참조서*의 내용을 참조하십시오. IMPORT, LOAD, RESTORE 유틸리티에 대해서는 *관리 안내서*의 내용을 참조하십시오.

다음 SQL문은 임의로 생성된 테스트 데이터로 사용자 테이블을 상주시키는 데 사용할 수 있는 기술을 보여줍니다. 다음 CREATE TABLE문에서처럼 EMP 테이블에 네 개의 컬럼, 즉 ENO(사원 번호), LASTNAME(성), HIREDATE(고용일), SALARY(사원 급여액)가 포함되어 있습니다.

```
  CREATE TABLE EMP (ENO INTEGER, LASTNAME VARCHAR(30),
                    HIREDATE DATE, SALARY INTEGER);
```

이 테이블을 1에서 어떤 숫자, 예를 들면 100까지의 사원 번호로 채우고 나머지 컬럼은 임의의 데이터로 채운다고 가정하십시오. 다음 SQL문을 사용하여 이를 수행할 수 있습니다.

```
INSERT INTO EMP
-- generate 100 records
WITH DT(ENO) AS (VALUES(1) UNION ALL
SELECT ENO+1 FROM DT WHERE ENO < 100 )  1

-- Now, use the generated records in DT to create other columns
-- of the employee record.
SELECT ENO,  2
    TRANSLATE(CHAR(INTEGER(RAND()*1000000)),  3
              CASE MOD(ENO,4) WHEN 0 THEN 'aeiou' || 'bcdfg'
                              WHEN 1 THEN 'aeiou' || 'hjklm'
                              WHEN 2 THEN 'aeiou' || 'npqrs'
                                     ELSE 'aeiou' || 'twxyz' END,
                                          '1234567890') AS LASTNAME,
    CURRENT DATE - (RAND()*10957) DAYS AS HIREDATE,  4
    INTEGER(10000+RAND()*200000) AS SALARY  5
  FROM DT;

SELECT * FROM EMP;
```

다음은 위 명령문에 대한 설명입니다.

1. INSERT문의 첫번째 부분은 순환 부속 조회를 사용하여 처음 100명의 사원에 대해 100개 레코드를 생성하여 사원 번호를 작성합니다. 각 레코드에는 사원 번호가 포함되어 있습니다. 사원 번호를 변경하려면 100이 아닌 번호를 사용하십시오.

2. SELECT문은 LASTNAME 컬럼을 생성합니다. RAND 함수를 사용하여 최고 6자리의 임의의 정수를 작성하여 시작됩니다. 그런 후 CHAR 함수를 사용하여 정수를 숫자 형식으로 변환시킵니다.
3. 숫자를 영문자로 변환하기 위해 이 명령문은 TRANSLATE 함수를 사용하여 10개의 숫자(0-9)를 영문자로 변환시킵니다. 10개 이상의 영문자가 있으므로 이 명령문은 다섯 개의 서로 다른 변환으로부터 선택합니다. 그 결과 발음할 수 있을 만큼의 충분한 수의 모음을 가진 이름이 되므로, 각 변환에 모음이 포함됩니다.
4. 명령문은 임의의 HIREDATE 값을 생성합니다. HIREDATE 값은 현재 날짜를 30년 전으로 다시 범위를 조정합니다. HIREDATE는 현재 날짜로부터 0에서 10 957 사이의 날짜를 빼서 계산됩니다.(10 957은 30년 동안의 날짜 수입니다.)
5. 마지막으로 명령문은 임의로 SALARY를 생성합니다. 최소 급여액은 10 000이고, 여기에 0에서 200 000 사이의 임의 숫자가 추가됩니다.

임의의 테스트 데이터를 생성하는 데 도움이 되는 샘플 프로그램을 보려면 `sqllib/samples/c` 서브디렉토리에 있는 `fillcli.sqc` 및 `fillsrv.sqc` 샘플 프로그램을 참조하십시오.

테스트 데이터에 대해 개발 중인 사용자 정의 함수(UDF)의 프로토타입을 고려해 볼 수도 있습니다. UDF 작성 이유 및 방법에 대한 세부사항은 433 페이지의 『제15장 사용자 정의 함수(UDF)』 및 409 페이지의 『제14장 사용자 정의 함수(UDFs)와 메소드』의 내용을 참조하십시오.

사용자 프로그램 수행, 테스트 및 디버깅

*응용프로그램 빌드 안내서*에는 사용자 환경에서 프로그램을 수행하는 방법을 설명합니다. 코드의 테스트 및 디버깅하는 동안 다음을 수행할 수 있습니다.

- 50 페이지의 『SQL문 프로토타입』에 설명된 동일한 기술을 사용하십시오. 여기에는 명령행 처리기 사용, Explain 기능, 사용자 프로그램이 조작 중인 데이터베이스 및 테이블에 대한 정보를 위한 시스템 카탈로그 뷰 분석, 생산 조건을 시뮬레이트하기 위한 특정 시스템 카탈로그 통계 갱신 등이 포함됩니다.

- 데이터베이스 시스템 모니터을 이용하여 분석할 최적화 정보를 캡처합니다. *시스템 모니터 안내 및 참조서*에서 자세한 내용을 참조하십시오.
- 플래거 기능을 사용하여 OS/390용 DB2 Universal Database용으로, 또는 SQL92 Entry Level 표준에 일치시키기 위한 목적으로 개발 중인 응용프로그램에 있는 SQL문 구문을 점검하십시오. 이 기능은 사전 처리 컴파일 중에 호출됩니다. 그 방법에 대한 정보는 해당 절의 끝부분에 있는 60 페이지의『사전 처리 컴파일』의 내용을 참조하십시오.
- 오류 처리 API를 전면 이용하십시오. 예를 들어, 테스트 단계 중에 모든 메시지를 인쇄하기 위해 오류 처리 API를 사용할 수 있습니다. 오류 처리 API에 대한 세부사항은 *Administrative API Reference*의 내용을 참조하십시오.

SQL문 프로토타입

사용자 응용프로그램을 설계 및 코드화할 때 특정 데이터베이스 관리 프로그램 기능 및 유틸리티를 이용하여 SQL 코드의 일부를 프로토타입하고 성능을 향상시킬 수 있습니다. 예를 들어, 다음을 수행할 수 있습니다.

- 명령행 처리기(CLP)을 사용하여 많은 SQL문을 테스트한 후 완료 프로그램을 컴파일 및 링크하십시오.

  그 결과 데이터베이스 테이블, 색인 또는 뷰에 저장된 정보를 정의 및 조작할 수 있습니다. 테이블 목차로부터 보고서를 생성하는 것 외에도 정보를 추가, 삭제 또는 갱신할 수 있습니다. 사용자의 Embedded SQL 프로그램에서 호스트 변수를 사용하려면 일부 SQL문에 대해 구문을 최소한으로 변경해야 합니다. 호스트 변수는 사용자 화면으로 출력되는 데이터를 저장하는 데 사용됩니다. 또한, 일부 Embedded SQL문(예. BEGIN DECLARE SECTION)은 해당 환경과 관련이 없으므로 CLP에 의해 지원되지 않습니다. CLP에 의해 지원되지 않는 SQL문이 무엇인지 보려면 829 페이지의 표37의 내용을 참조하십시오.

  또한 명령행 처리기 요청의 입력 및 출력을 경로 재지정할 수도 있습니다. 예를 들어, 명령행 처리기 요청으로의 입력으로서 필요한 SQL문이 포함된 하나 이상의 파일을 작성하여 명령문의 재입력을 저장할 수 있습니다.

  명령행 처리기에 대해서는 *Command Reference*의 내용을 참조하십시오. 명령 센터에 대해서는 *관리 안내서*의 내용을 참조하십시오.

- Explain 기능을 사용하여 프로그램에 사용할 DELETE, INSERT, UPDATE 또는 SELECT문의 예상 비용을 알아 보십시오. Explain 기능은 subject문의 예상 비용 및 구조에 대한 정보를 사용자 제공 테이블에 둡니다. Visual Explain 또는 `db2exfmt` 유틸리티를 사용하여 이 정보를 볼 수 있습니다.

  Explain 기능 사용 방법에 대해서는 *관리 안내서: 구현*의 내용을 참조하십시오.

- 시스템 카탈로그 뷰를 사용하여 기존의 데이터베이스에 대한 정보를 쉽게 검색할 수 있습니다. 데이터베이스 관리 프로그램은 데이터베이스가 작성, 교체 및 갱신될 때 정상 조작 중에 뷰가 기초하고 있는 시스템 카탈로그 테이블을 작성 및 유지보수합니다. 이 뷰에는 권한 부여된 권한, 컬럼 이름, 데이터 유형, 색인, 패키지 종속성, 참조 제한조건, 테이블 이름, 뷰 등의 각 데이터베이스에 대한 데이터가 들어 있습니다. 시스템 카탈로그의 데이터는 일반 SQL 조회 기능을 통해 사용 가능합니다.

  SQL 최적화 알고리즘에 의해 사용되는 통계 정보가 포함된 일부 시스템 카탈로그 뷰를 갱신할 수 있습니다. 가설 데이터베이스 성능을 조사하거나 최적화 알고리즘에 영향을 주기 위해 이 뷰의 일부 컬럼을 변경할 수 있습니다. 이 방법을 사용하여 개발중의 생산 시스템을 시뮬레이트하고 시스템을 테스트하며, 조회 수행 방식을 분석할 수 있습니다.

  각 시스템 카탈로그 뷰에 대한 전체 설명은 *SQL 참조서*의 부록을 참조하십시오.시스템 카탈로그 통계 및 변경할 통계에 대해서는 *관리 안내서: 구현*의 내용을 참조하십시오.

제2부 응용프로그램에서의 Embedding SQL

© Copyright IBM Corp. 1993, 2000

제3장 Embedded SQL 개요

호스트 언어에서의 Embedding SQL문 . . . 55
소스 파일 작성 및 준비 57
Embedded SQL용 패키지 작성 60
  사전 처리 컴파일 60
    소스 파일 요구사항 62
  컴파일 및 링크 64
  바인딩 65
    패키지 이름 바꾸기 65
    동적 명령문 바인딩 66
    규정화되지 않는 테이블 이름 분석 . . 67
    기타 바인딩 고려사항 67
지연 바인딩의 이점 69
DB2 바인드 파일 설명 유틸리티 - db2bfd 69
응용프로그램, 바인드 파일 및 패키지 관계 70
시간소인 71
리바인딩 72

호스트 언어에서의 Embedding SQL문

호스트 언어 안에 포함된 SQL문으로 응용프로그램을 작성할 수 있습니다. SQL 문은 데이터베이스 인터페이스를 제공하는 반면, 호스트 언어는 응용프로그램이 실행되는 데 필요한 나머지 지원을 제공합니다.

표2에는 호스트 언어 응용프로그램에 내포된 SQL문이 있습니다. 이 예에서 응용프로그램은 SQLCA 구조의 SQLCODE 필드를 점검하여 갱신이 성공했는지를 결정합니다.

표 2. 호스트 언어에서의 Embedding SQL문

언어	샘플 소스 코드

C/C++	`EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr';`
	`if ( SQLCODE < 0 )`
	`   printf( "Update Error:  SQLCODE = %ld \n", SQLCODE );`

Java (SQLJ)	`try {`
	`   #sql { UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr' };`
	`}`
	`catch (SQLException e) {`
	`   println( "Update Error:  SQLCODE = " + e.getErrorCode() );`
	`}`

COBOL	`EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr' END_EXEC.`
	`IF SQLCODE LESS THAN 0`
	`   DISPLAY 'UPDATE ERROR:  SQLCODE = ', SQLCODE.`

© Copyright IBM Corp. 1993, 2000

표 2. 호스트 언어에서의 Embedding SQL문 (계속)

언어	샘플 소스 코드

FORTRAN	`EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr'`
	`if ( sqlcode .lt. 0 ) THEN`
	`   write(*,*) 'Update error:  sqlcode = ', sqlcode`

응용프로그램에 있는 SQL문은 호스트 언어마다 다르지 않습니다. 데이터베이스 관리 프로그램은 호스트 언어에 의해 처리되는 SQL 구문 변환 방식을 제공합니다.

C, C++, COBOL 또는 FORTRAN 언어의 경우, 이러한 변환은 DB2 사전 처리 컴파일러에 의해 처리됩니다. DB2 사전 처리 컴파일러는 PREP 명령을 사용하여 호출됩니다. 사전 처리 컴파일러는 Embedded SQL문을 직접 DB2 런타임 서비스 API 호출로 변환시킵니다.

Java 언어의 경우 SQLJ 변환기는 SQLJ절을 JDBC문으로 변환합니다. SQLJ 변환기는 SQLJ 명령으로 호출합니다.

사전 처리 컴파일러가 소스 파일을 처리할 때 SQL문을 검색하고 비-SQL 호스트 언어를 삼가합니다. SQL문은 특수 분리문자에 의해 둘러싸여 있으므로 쉽게 찾을 수 있습니다. 사용 중인 언어로 SQL문을 내포시키는 데 필요한 구문 정보는 다음을 보십시오.

- C/C++용, 668 페이지의 『C 및 C++에 SQL문 포함』
- Java (SQLJ)용, 730 페이지의 『Java에 SQL문 포함』
- COBOL용, 765 페이지의 『COBOL에서의 Embedding SQL문』
- FORTRAN용, 793 페이지의 『FORTRAN에서의 SQL문 포함』
- REXX용, 808 페이지의 『REXX에서 Embedding SQL문』

표3에는 지원되는 컴파일 호스트 언어로 유효한 Embedded SQL문을 작성하기 위해 분리문자 및 주석을 사용하는 방법을 보여 줍니다.

표 3. 호스트 언어에서의 Embedding SQL문

언어	샘플 소스 코드
C/C++	/* Only C or C++ comments allowed here */ EXEC SQL -- SQL comments or /* C comments or */ // C++ comments allowed here DECLARE C1 CURSOR FOR sname; /* Only C or C++ comments allowed here */

SQLJ	/* Only Java comments allowed here */
	#sql c1 = {
	-- SQL comments or
	/* Java comments or */
	// Java comments allowed here
	SELECT name FROM employee
	};
	/* Only Java comments allowed here */

COBOL	* See COBOL documentation for comment rules * Only COBOL comments are allowed here EXEC SQL -- SQL comments or * full-line COBOL comments are allowed here DECLARE C1 CURSOR FOR sname END-EXEC. * Only COBOL comments are allowed here
FORTRAN	C Only FORTRAN comments are allowed here EXEC SQL + -- SQL comments, and C full-line FORTRAN comment are allowed here + DECLARE C1 CURSOR FOR sname I=7 ! End of line FORTRAN comments allowed here C Only FORTRAN comments are allowed here

소스 파일 작성 및 준비

텍스트 편집기를 사용하여, 소스 파일이라는 표준 ASCII 파일에 소스 코드를 작성할 수 있습니다. 소스 파일에는 코드를 작성하는 호스트 언어에 적절한 확장자가 있어야 합니다. 사용 중인 호스트 언어에 대해 필요한 파일 확장자를 알아보려면 837 페이지의 표38의 내용을 참조하십시오.

주: 모든 플랫폼이 모든 호스트 언어를 지원하는 것은 아닙니다. 구체적인 정보는 *응용프로그램 빌드 안내서*의 내용을 보십시오.

여기에서는 이미 소스 코드를 작성했다고 가정합니다.

컴파일된 호스트 언어를 사용하여 응용프로그램을 작성한 경우 추가 단계를 따라 응용프로그램을 작성해야 합니다. 프로그램 컴파일 및 링크를 사용하여 *사전 처리 컴파일* 및 *바인드*를 수행해야 합니다.

간단히 말해서 사전 처리 컴파일은 Embedded SQL문을 호스트 컴파일러가 처리할 수 있는 DB2 런타임 API로 변환시키고 바인드 파일을 작성합니다. 바인드 파일에는 응용프로그램에 있는 SQL문에 대한 정보가 들어 있습니다. BIND 명령은 데이터베이스에 *패키지*를 작성합니다. 선택적으로 사전 처리 컴파일러는 사전 처리 컴파일시 바인드 단계를 수행할 수 있습니다.

바인딩은 바인드 파일로부터 *패키지*를 작성하고, 이를 데이터베이스에 저장하는 프로세스입니다. 사용자 응용프로그램이 한 개 이상의 데이터베이스에 액세스할 경우 각 데이터베이스에 대해 하나의 패키지를 작성해야 합니다.

59 페이지의 그림1에는 일반 컴파일된 DB2 응용프로그램의 다양한 모듈과 함께 이러한 단계 순서가 나와 있습니다. 프로그램 확장의 각 단계마다 발생하는 사항에 대한 다음 절을 읽어 가면서 이를 참조하고자 할 수 있습니다.

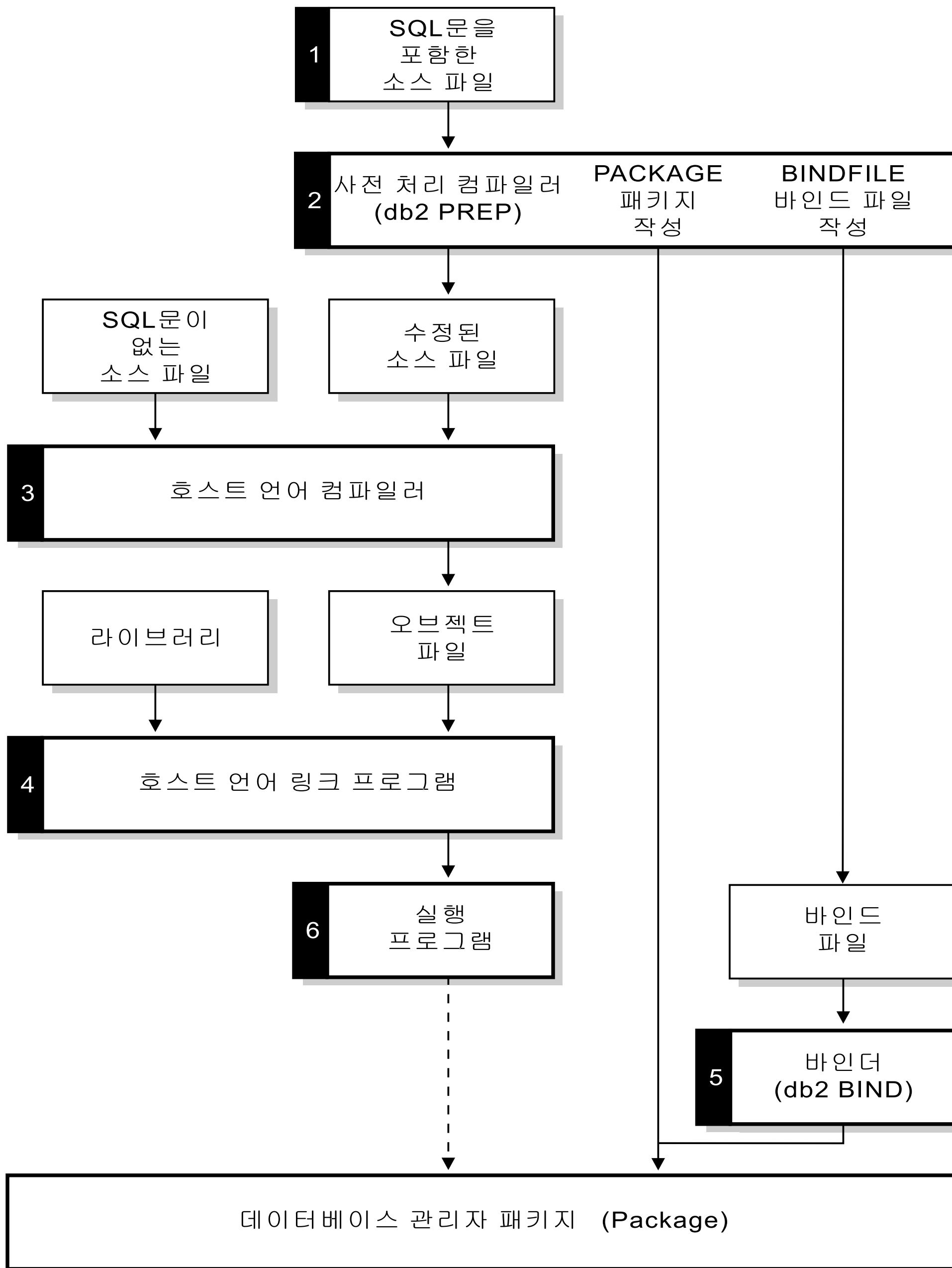


그림 1. 컴파일된 호스트 언어로 작성된 프로그램 준비

Embedded SQL용 패키지 작성

컴파일된 호스트 언어로 작성된 응용프로그램을 수행하려면 실행시 데이터베이스 관리 프로그램에 필요한 패키지를 작성해야 합니다. 여기에는 59 페이지의 그림1에 표시된 바와 같이 다음 단계가 포함됩니다.

- 사전 처리 컴파일(2단계), Embedded SQL 소스문을 데이터베이스 관리 프로그램이 사용할 수 있는 형식으로 변환합니다.
- 컴파일 및 링크(3-4단계), 필요한 오브젝트 모듈을 작성합니다.
- 바인딩(5단계), 프로그램이 수행될 때 데이터베이스 관리 프로그램이 사용할 패키지를 작성합니다.

이 절에서 논의되는 기타 주제는 다음과 같습니다.

- 응용프로그램, 바인드 파일 및 패키지 관계,
- 리바인딩, 패키지 리바인드 시기 및 방법에 대해 설명합니다.

SQLJ 응용프로그램에 필요한 패키지를 작성하려면 SQLJ 변환기와 db2profc 명령 모두를 사용해야 합니다. SQLJ 변환기 사용에 대해서는 726 페이지의『SQLJ 프로그래밍』에서 자세한 내용을 참조하십시오.

사전 처리 컴파일

소스 파일을 작성한 후, 호스트 언어 소스 파일에 대해 PREP 명령으로 SQL문이 포함된 각 호스트 언어 파일을 사전 처리 컴파일해야 합니다. 사전 처리 컴파일러는 소스 파일에 포함된 SQL문을 주석으로 변환하고, 이러한 명령문에 대한 DB2 런타임 API 호출을 생성합니다.

응용프로그램을 사전 처리 컴파일하기 전에 서버에 내재적 또는 명시적으로 연결해야 합니다. 클라이언트 워크스테이션에서 응용프로그램을 사전 처리 컴파일하고 사전 처리 컴파일러가 클라이언트에서 수정된 소스 및 메시지를 생성해도, 사전 처리 컴파일러는 검증을 수행하기 위해 서버 연결을 사용합니다.

사전 처리 컴파일러는 데이터베이스 관리 프로그램이 데이터베이스에 대해 SQL문을 처리하는 데 필요한 정보도 작성합니다. 이 정보는 선택된 사전 처리 컴파일러 옵션에 따라 패키지, 바인드 파일 또는 두 파일 모두에 저장됩니다.

사전 처리 컴파일러 사용의 일반적인 예는 다음과 같습니다. *filename.sqc*이라는 C Embedded SQL 소스 파일을 사전 처리 컴파일하기 위해, 기본 이름이 `filename.c`인 C 소스 파일과 기본 이름이 `filename.bnd`인 바인드 파일을 작성하기 위해 다음 명령을 발행할 수 있습니다.

```
DB2 PREP filename.sqc BINDFILE
```

사전 처리 컴파일러 구문 및 옵션에 대한 세부사항은 *Command Reference*의 내용을 참조하십시오.

사전 처리 컴파일러는 당과 같은 네개의 유형을 생성합니다.

- 수정된 소스
- 패키지
- 바인드 파일
- 메시지 파일

수정된 파일 이 파일은 사전 처리 컴파일러가 SQL문을 DB2 런타임 API 호출로 변환시킨 후 새로운 버전의 원래 소스 파일입니다. 적절한 호스트 언어 확장이 제공됩니다.

패키지 PACKAGE 옵션(기본값)을 사용하거나 BINDFILE, SYNTAX 또는 SQLFLAG 옵션을 지정하지 않으면 패키지는 연결된 데이터베이스에 저장됩니다. 패키지에는 이 데이터베이스에 대해서만 특정 소스 파일의 정적 SQL문을 실행하는 데 필요한 모든 정보가 들어 있습니다. PACKAGE USING 옵션과 다른 이름을 지정하지 않는 한 사전 처리 컴파일러는 소스 파일 이름의 첫번째 8자로부터 패키지 이름을 형성합니다.

PACKAGE 옵션을 이용하여 사전 처리 컴파일 프로세스 중에 사용된 데이터베이스에는 소스 파일에 있는 정적 SQL문에 의해 참조되는 모든 데이터베이스 오브젝트가 포함되어야 합니다. 예를 들어, 참조되는 테이블이 데이터베이스에 있지 않는 한 SELECT문을 사전 처리 컴파일할 수 없습니다.

바인드 파일 BINDFILE 옵션을 사용하면, 사전 처리 컴파일러는 패키지를 작성하는 데 필요한 데이터가 포함된 바인드 파일(확장자는 `.bnd`)

을 작성합니다. 나중에 BIND 명령과 함께 이 파일을 사용하여 여러 데이터베이스로 응용프로그램을 바인드할 수 있습니다. BINDFILE을 지정하고 PACKAGE 옵션을 지정하지 않는 한, BIND 명령을 호출할 때까지 바인딩이 지연됩니다. CLP(Command Line Processor)의 경우 PREP의 기본값은 BINDFILE 옵션을 지정하지 않습니다. 따라서, CLP를 사용중이거나 바인딩을 지연시키려는 경우 BINDFILE 옵션을 지정해야 합니다.

사전 처리 컴파일시 바인드 파일을 요청하지만 PACKAGE를 지정하지 않으면, 즉 패키지를 작성하지 않으면 일부 오브젝트의 존재 및 권한 부여 SQLCODE는 오류가 아닌 경고로 취급됩니다. 그 결과 참조된 오브젝트 없이도, 사전 처리 컴파일 중인 SQL문 실행 권한 없이도 사용자는 바인드 파일을 작성할 수 있고 프로그램을 사전 처리 컴파일할 수 있습니다. 오류가 아닌 경고로 취급되는 특정 SQLCODE 목록은 *Command Reference*의 내용을 참조하십시오.

메시지 파일 MESSAGES 옵션을 사용할 경우 사전 처리 컴파일러는 메시지를 표시된 파일로 경로 재지정합니다. 이러한 메시지에는 사전 처리 컴파일 중에 발생할 수 있는 문제점을 기술하는 오류 및 경고 메시지가 포함됩니다. 소스 파일이 성공적으로 사전 처리 컴파일되지 않는 경우, 경고 및 오류 메시지를 사용하여 문제점을 판별하고, 소스 파일을 정정한 후, 다시 소스 파일을 사전 처리 컴파일해 보십시오. MESSAGES 옵션을 사용하지 않으면 사전 처리 컴파일 메시지가 표준 출력으로 기록됩니다.

소스 파일 요구사항

응용프로그램 내에서 데이터베이스를 사용하지 않아도, 특정 데이터베이스에 대해 항상 소스 파일을 사전 처리 컴파일해야 합니다. 사실상, 개발용 테스트 데이터베이스를 사용하고, 완전히 테스트를 한 후에는 그 바인드 파일을 하나 이상의 제품 데이터베이스를 바인드할 수 있습니다. 이 기능을 사용할 다른 방법은 69 페이지의 『지연 바인딩의 이점』의 내용을 참조하십시오.

사용자 응용프로그램이 데이터베이스 코드 페이지와 다른 코드 페이지를 사용할 경우, 사전 처리 컴파일시 어떤 코드 페이지를 사용할 것인지 고려해야 합니다. 573 페이지의 『다른 코드 페이지 간의 변환』에서 자세한 내용을 참조하십시오.

응용프로그램이 사용자 정의 함수(UDF) 또는 사용자 정의 구별 유형(UDT)을 사용할 경우, 응용프로그램을 사전 처리 컴파일할 때 FUNCPATH 옵션을 사용해야 합니다. 이 옵션은 정적 SQL이 포함된 응용프로그램에 대해 UDF 및 UDT를 분석하는 데 사용되는 함수 경로를 지정합니다. FUNCPATH가 지정되지 않을 경우 기본 함수 경로는 *SYSIBM*, *SYSFUN*, *USER*입니다. 여기서 *USER*는 현재 사용자 ID를 참조합니다. 바인드 옵션에 대한 세부사항은 *Command Reference*의 내용을 참조하십시오.

하나 이상의 서버에 액세스하는 응용프로그램을 사전 처리 컴파일하려면 다음을 수행할 수 있습니다.

- 각 데이터베이스에 대한 SQL문을 별도의 소스 파일로 분할하십시오. 같은 파일에서 다른 데이터베이스용 SQL문을 혼합하지 마십시오. 각 소스 파일은 적절한 데이터베이스에 대해 사전처리 컴파일될 수 있습니다. 이것이 추천 메소드입니다.
- 동적 SQL문만 사용하여 응용프로그램을 코드화하고, 플그램이 액세스할 각 데이터베이스에 대한 바인드를 수행하십시오.
- 모든 데이터베이스가 같아 보이면, 즉 정의가 같으면, SQL문들을 하나의 소스 파일로 그룹화할 수 있습니다.

동일한 프로시듀어는 응용프로그램이 DB2 Connect을 통해 호스트나 AS/400 응용프로그램 서버를 액세스할 때 적용됩니다. 해당 서버에 대해 사용 가능한 PREP 옵션을 사용하여 연결할 서버에 대해 사전 처리 컴파일하십시오.

OS/390용 DB2 Universal Database에서 수행될 응용프로그램을 사전 처리 컴파일할 경우, SQL문 구문을 점검할 때 플래거 기능을 사용해 보십시오. 플래거에는 DB2 Universal Database가 지원하지만 OS/390용 DB2 Universal Database는 지원하지 않는 SQL 구문을 나타냅니다. SQL 구문이 SQL92 Entry Level 구문을 따르는지 점검할 때에도 플래거를 사용할 수 있습니다. PREP 명령에서 SQLFLAG 옵션을 사용하여 호출하고, 비교에 사용할 OS/390용 DB2 Universal Database SQL 구문 버전을 지정할 수 있습니다. 플래거 기능은 SQL 사용시 변

경을 강화하지 않습니다. 이 기능은 구문 비호환성과 관련한 정보 및 경고 메시지만 발행하고, 선행 처리는 비정상적으로 종료하지 않습니다.

PREP 명령에 대한 세부사항은 *Command Reference*의 내용을 참조하십시오.

컴파일 및 링크

수정된 소스 파일 및 적절한 호스트 언어 컴파일러를 사용하는 SQL문이 포함되지 않은 추가 소스 파일을 컴파일하십시오. 언어 컴파일러는 수정된 각 소스 파일을 *오브젝트* 모듈로 변환합니다.

기본 컴파일 옵션의 예외는, 사용자 운영 플랫폼에 대한 기타 프로그래밍 문서 또는 *응용프로그램 빌드 안내서*의 내용을 참조하십시오. 사용 가능한 컴파일 옵션에 대한 전체 설명은 컴파일러 문서를 참조하십시오.

호스트 언어 링커는 실행 응용프로그램을 작성합니다. 예를 들면 다음과 같습니다.
- OS/2 및 Windows 32비트 운영 체제에서 응용프로그램은 실행 파일 또는 동적 링크 라이브러리(DLL)이 될 수 있습니다.
- UNIX 기반 시스템에서, 응용프로그램은 실행가능 로드 모듈 또는 공유 라이브러리가 될 수 있습니다.

주: 응용프로그램이 Windows 32비트 운영 체제에서 DLL이 될 수 있어도 DLL은 응용프로그램에 의해 직접 로드되며 데이터베이스 관리 프로그램에 의해서는 로드되지 않습니다. Windows 32 비트 운영 체제에서는 데이터베이스 관리 프로그램은 DLL을 로드할 수 있습니다. 일반적으로 저장 프로시듀어는 DLL 또는 공유 라이브러리로서 구축됩니다. 저장 프로시듀어 사용에 대해서는 207 페이지의『제7장 저장 프로시듀어』의 내용을 참조하십시오.

DB2에 의해 지원되는 기타 플랫폼에서의 실행 파일 작성에 대해서는 *응용프로그램 빌드 안내서*의 내용을 참조하십시오.

실행 파일을 작성하려면 다음을 링크하십시오.
- SQL문이 포함되지 않은 기타 파일 및 수정된 소스 파일로부터의 언어 컴파일러에 의해 생성된 사용자 오브젝트 모듈
- 언어 컴파일러에서 제공되는 호스트 언어 라이브러리 API

- 운영 환경에 대한 데이터베이스 관리 프로그램 API가 포함된 데이터베이스 관리 프로그램 라이브러리. 데이터베이스 관리 프로그램 API에 필요한 데이터베이스 관리 프로그램 라이브러리의 구체적인 이름은, *응용프로그램 빌드 안내서* 또는 사용자의 운영 플랫폼에 대한 기타 프로그래밍 문서를 참조하십시오.

바인딩

바인딩이란 응용프로그램이 실행될 때 데이터베이스에 액세스하기 위해 데이터베이스 관리 프로그램이 필요로 하는 패키지를 작성하는 프로세스를 말합니다. 바인딩은 사전 처리 컴파일 중에 PACKAGE 옵션을 지정하여 내재적으로 수행되거나, 사전 처리 컴파일 중에 작성되는 바인드 파일에 대해 BIND 명령을 사용하여 명시적으로 수행될 수 있습니다.

이어서 BIND 명령 사용의 일반적인 예가 나옵니다. *filename.bnd*라는 바인드 파일을 데이터베이스로 바인드하려면 다음 명령을 발행하십시오.

```
DB2 BIND filename.bnd
```

BIND 명령 구문 및 옵션에 대한 세부사항은 *Command Reference*의 내용을 참조하십시오.

별도로 사전 처리 컴파일된 소스 코드 모듈에 대해 패키지 하나가 작성됩니다. 응용프로그램 하나에 다섯 개의 소스 파일이 있고, 이 중 세 파일에 사전 처리 컴파일이 필요할 경우, 세 개의 패키지 또는 바인드 파일이 작성됩니다. 기본적으로 각 패키지에는 .bnd 파일의 출처가 되는 소스 모듈 이름과 같은 이름이 제공되며, 그 이름은 8자로 절단됩니다. 이 새로 작성된 패키지 이름이 현재 목표 데이터베이스에 있는 패키지와 같은 경우, 새 패키지는 이전의 패키지를 대체합니다. 다른 패키지 이름을 명시적으로 지정하려면 PREP 명령에서 PACKAGE USING 옵션을 사용해야 합니다. *Command Reference*에서 자세한 내용을 참조하십시오.

패키지 이름 바꾸기

여러 버전의 응용프로그램을 작성할 때 패키지를 재명명하여 이름 충돌을 피하십시오. 예를 들어, (foo.sqc로부터 컴파일된) foo라는 응용프로그램이 있는 경우 이를 사전 처리 컴파일한 후 응용프로그램의 모든 사용자에게 보냅니다. 사용자는 응용프로그램을 데이터베이스에 바인드한 후, 응용프로그램을 수행합니다. 이후 변경을 하려면, 새로운 버전의 foo를 작성하고 이 응용프로그램 및 바인드 파일을

새로운 버전을 필요로 하는 사용자에게 보내십시오. 새로운 사용자는 foo.bnd를 바인드하고 새 응용프로그램은 문제 없이 수행됩니다. 그러나, 사용자가 구 버전의 응용프로그램을 수행할 때, FOO 패키지에서의 시간소인 충돌이 발생합니다.(이는 데이터베이스의 패키지가 수행 중인 응용프로그램과 일치하지 않음을 나타냅니다.) (패키지 시간소인에 대해서는 71 페이지의 『시간소인』의 내용을 참조하십시오.) 이제 새 응용프로그램 사용자는 시간소인 충돌을 받습니다. 이러한 문제는 두 응용프로그램 모두 같은 이름의 패키지를 사용하므로 발생합니다.

해결 방안은 패키지 이름을 바꾸는 것입니다. 첫번째 버전의 FOO를 구축할 때 다음 명령으로 사전 처리 컴파일해야 합니다.

```
DB2 PREP FOO.SQC BINDFILE PACKAGE USING FOO1
```

이 응용프로그램을 분산할 경우 사용자는 문제없이 이를 바인드 및 수행할 수 있습니다. 새 버전을 구축할 때 다음 명령으로 사전 처리 컴파일합니다.

```
DB2 PREP FOO.SQC BINDFILE PACKAGE USING FOO2
```

새로운 응용프로그램을 분산한 후에는 문제 없이 바인드 및 수행됩니다. 새 버전의 패키지 이름이 FOO2이고 첫번째 버전의 패키지 이름이 FOO1이므로, 이름 충돌은 없고 두 버전의 응용프로그램을 사용할 수 있습니다.

동적 명령문 바인딩

동적으로 준비된 명령문의 경우 특수 레지스터 값들이 명령문 컴파일 환경을 결정합니다.

- CURRENT QUERY OPTIMIZATION 특수 레지스터는 어떤 최적화 클래스가 사용되는지 결정합니다.
- CURRENT FUNCTION PATH 특수 레지스터는 UDF 및 UDT 분석에 사용되는 함수 경로를 결정합니다.
- CURRENT EXPLAIN SNAPSHOT 레지스터는 Explain 스냅샷 정보가 캡쳐되는지 결정합니다.
- CURRENT EXPLAIN MODE 레지스터는 사용가능한 동적 SQL문에 대해 Explain 테이블 정보가 캡쳐되는지 결정합니다. 이 특수 레지스터의 기본값은 관련 바인드 옵션에 사용되는 것과 기본값이 같습니다. 특수 레지스터 및 BIND 옵션과의 상호작용에 대해서는 *SQL 참조서* 부록을 참조하십시오.

규정화되지 않는 테이블 이름 분석

다음 방법을 사용하여 응용프로그램에 있는 규정화되지 않은 테이블 이름을 처리할 수 있습니다.

- 사용자 각각에 대해, 다음 명령을 사용하여 다른 권한 부여 식별자와 다른 COLLECTION 매개변수를 가진 패키지를 바인드하십시오.

  ```
  CONNECT TO db_name USER user_name
  BIND file_name COLLECTION schema_name
  ```

  위 예에서, *db_name*은 데이터베이스 이름이며, *user_name*은 사용자 이름이고, *file_name*은 바인드될 응용프로그램 이름입니다. *user_name* 및 *schema_name*은 항상 같은 값입니다. 그런 후, SET CURRENT PACKAGESET문을 사용하여 어떤 패키지를 사용하고, 따라서 어떤 규정자를 사용할 것인지 지정하십시오. 기본 규정자는 패키지를 바인딩할 때 사용되는 권한 부여 식별자입니다. SET CURRENT PACKAGESET문 사용 방법의 예는 *SQL 참조서*의 내용을 참조하십시오.
- 테이블과 같은 이름을 가진 각 사용자에 대한 뷰를 작성하여 규정화되지 않은 테이블 이름이 제대로 되도록 하십시오.(QUALIFIER 옵션은 DB2 Connect 전용이고, 호스트 서버를 사용할 때만 사용할 수 있음을 의미합니다.)
- 각 사용자가 원하는 테이블을 가리키는 별명을 작성하십시오.

기타 바인딩 고려사항

사용자 응용프로그램 코드 페이지가 데이터베이스 코드 페이지와 다른 코드 페이지를 사용할 경우, 바인딩할 때 어떤 코드 페이지를 사용할 것인지 고려해야 합니다. 573 페이지의 『다른 코드 페이지 간의 변환』에서 자세한 내용을 참조하십시오.

응용프로그램이 IMPORT 또는 EXPORT와 같은 데이터베이스 관리 프로그램 유틸리티 API에 대한 호출을 발행하는 경우, 제공된 유틸리티 바인드 파일을 데이터베이스로 바인드해야 합니다. 세부사항은 사용자 플랫폼용 *빠른 시작* 안내서를 참조하십시오.

바인드 옵션을 사용하여 다음 예에서처럼 바인딩 중에 발생하는 일부 조작을 제어할 수 있습니다.

- QUERYOPT 바인드 옵션은 바인딩될 때 특정 최적화 클래스를 이용합니다.

- EXPLSNAP 바인드 옵션은 Explain 테이블에 사용가능한 SQL문에 대한 Explain 스냅샷 정보를 저장합니다.
- FUNCPATH 바인드 옵션은 사용자 정의 구별 유형 및 사용자 정의 함수를 정적 SQL에서 제대로 분석됩니다.

바인드 옵션에 대한 세부사항은 *Command Reference*에 있는 BIND 명령 관련 절을 참조하십시오.

바인드 프로세스가 시작되지만 리턴되지 않는 경우, 데이터베이스에 연결된 기타 응용프로그램이 사용자가 필요로 하는 잠금을 보유해야 합니다. 이런 경우, 데이터베이스에 응용프로그램이 연결될 수 없습니다. 그런 경우, 서버에서의 모든 응용프로그램을 연결해제하면 바인드 프로세스가 계속됩니다.

응용프로그램이 DB2 Connect를 사용하여 서버에 액세스할 경우, 그 서버에 대해 사용할 수 있는 BIND 옵션을 사용할 수 있습니다. BIND 명령 및 옵션에 대한 세부사항은 *Command Reference*의 내용을 참조하십시오.

바인드 파일은 이전 버전의 DB2 Universal Database와 역호환됩니다. 혼합 레벨 환경에서, DB2는 데이터베이스 환경의 최저 레벨이 사용할 수 있는 함수만 사용할 수 있습니다. 예를 들어, V5.2 클라이언트는 V5.0 서버로 연결할 경우 클라이언트는 V5.0 함수만 사용할 수 있습니다. 바인드 파일이 데이터베이스 기능을 표현하므로 혼합 레벨 제한을 받게 됩니다.

더 낮은 레벨의 시스템에서 보다 높은 레벨의 바인드 파일을 리바인드해야 할 경우 다음을 수행할 수 있습니다.

- 보다 낮은 레벨의 DB2 응용프로그램 개발 클라이언트를 사용하여 보다 높은 레벨의 서버에 연결하고 낮은 레벨의 DB2 Universal Database 환경에 제공되고 바인드될 수 있는 바인드 파일을 작성하십시오.
- 낮은 레벨의 제품 환경에서 높은 레벨의 DB2 클라이언트를 사용하여, 테스트 환경에서 작성된 높은 레벨의 바인드 파일을 바인드하십시오. 높은 레벨의 클라이언트는 낮은 레벨의 서버에 적용되는 옵션만 전달합니다.

지연 바인딩의 이점

바인딩이 가능한 사전 처리 컴파일을 통해 응용프로그램은 사전 처리 컴파일 중에 사용된 데이터베이스에만 액세스할 수 있습니다. 그러나, 지연되는 바인딩으로 사전 처리 컴파일함으로써 응용프로그램은 많은 데이터베이스에 액세스할 수 있습니다. 각 파일에 대해 BIND 파일을 바인드할 수 있기 때문입니다. 이 응용프로그램 개발 방법은, 응용프로그램이 한 번만 사전 처리 컴파일되지만 응용프로그램이 언제라도 데이터베이스에 바인드될 수 있다는 점에서 내재적으로 보다 융통성이 있습니다.

실행 중에 BIND API를 사용함으로써 설치 프로시듀어의 일부로서 또는 관련된 모듈이 실행되기 전에 응용프로그램은 스스로 바인드됩니다. 예를 들어, 응용프로그램은 여러 개의 타스크를 수행할 수 있고, 이 중 하나만 SQL문을 사용해야 합니다. 응용프로그램이 SQL문이 필요한 타스크를 호출하고, 관련된 패키지가 없는 경우에만 자신을 데이터베이스로 바인드하도록 응용프로그램을 설계할 수 있습니다.

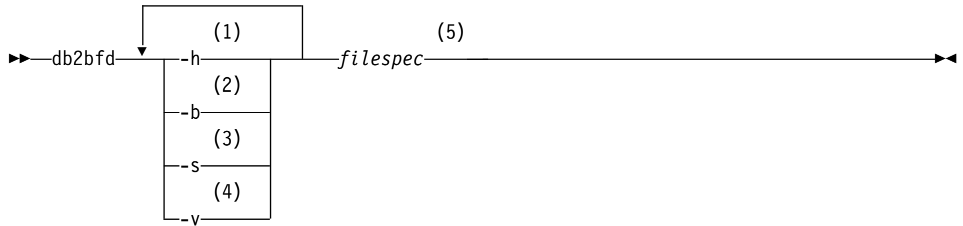
지연된 바인딩 방법의 또다른 이점은, 소스 코드를 일반 사용자에게 제공하지 않고 패키지를 작성할 수 있도록 한다는 점입니다. 응용프로그램과 함께 관련 바인드 파일을 제공할 수 있습니다.

DB2 바인드 파일 설명 유틸리티 - db2bfd

DB2 Bind File Description(`db2bfd`) 유틸리티를 이용하여 바인드 파일 작성에 사용되는 사전 처리 컴파일 옵션 뿐만 아니라 바인드 파일 내용을 쉽게 표시할 수 있습니다. 이를 통해 그 안에 있는 SQL문을 검토 및 검증하기가 쉬워집니다. 이는 응용프로그램의 바인드 파일과 관련된 문제점 판별에 유용할 수 있습니다.

`db2bfd` 유틸리티는 인스턴스의 `sqllib` 디렉토리의 `bin` 서브디렉토리에 있습니다.

Its 구문:



주:

1 도움말 정보 표시.

2 바인드 파일 헤더 표시.

3 SQL문 표시.

4 호스트 변수 선언 표시

5 바인드 파일의 이름.

db2bfd에 대해서는 *Command Reference*에서 자세한 내용을 참조하십시오.

응용프로그램, 바인드 파일 및 패키지 관계

패키지는 단일 소스 파일에서 특정 SQL문을 실행하는 데 필요한 정보가 포함된 데이터베이스에 저장된 오브젝트입니다. 데이터베이스 응용프로그램은 응용프로그램을 구축하는 데 사용되는 모든 사전 처리 컴파일 소스 파일에 대해 하나의 패키지를 사용합니다. 각 패키지는 분리되는 엔터티로서, 같은 응용프로그램 또는 기타 응용프로그램이 사용하는 기타 패키지와는 아무런 관련이 없습니다. 패키지는 바인딩이 작동되는 소스 파일에 대해 사전 처리 컴파일러를 수행하거나, 나중에 하나 이상의 바인드 파일을 이용하여 바인더를 수행함으로써 작성됩니다.

데이터베이스 응용프로그램은 응용프로그램이 컴파일되는 것과 같은 이유로, 즉 성능 향상 및 압축성 증가를 위해 패키지를 사용합니다. SQL문을 사전 처리 컴파일함으로써 응용프로그램이 수행될 때가 아닌 구축될 때 명령문이 패키지로 컴파일됩니다. 각 명령문이 분석되고, 보다 효율적으로 해석되는 피연산자 문자열이 패키지에 저장됩니다. 런타임시, 사전 처리 컴파일러가 생성하는 코드는 런타임 서비스 데이터베이스 관리 프로그램을 호출합니다. 입력 또는 출력 데이터에 필요한 변수 정보 및 패키지에 저장된 정보를 가진 API가 실행됩니다.

사전 처리 컴파일의 이점은 정적 SQL문에만 적용됩니다. (PREPARE 및 EXECUTE 또는 EXECUTE IMMEDIATE를 사용하여) 동적으로 실행되는 SQL문은 사전 처리 컴파일되지 않습니다. 따라서 런타임시 전체 처리 단계 세트를 통과해야 합니다.

주: SQL문의 정적 버전이 동적으로 처리되는 동일한 명령문보다 자동으로 빠르게 실행된다고 전제하지 마십시오. 동적 명령문 준비에 필요한 오버헤드로 인해 정적 SQL이 더 빠른 경우도 있습니다. 또는 최적화 알고리즘이 이전에 바인드될 때 사용 가능한 데이터베이스 통계가 아닌 현재 데이터베이스 통계를 사용할 수 있으므로, 동적으로 준비된 같은 명령문이 더 빨리 실행될 수도 있습니다. 트랜잭션 완료 시간이 2-3분 정도 덜 걸리는 경우 일반적으로 정적 SQL이 더 빠릅니다. 어떤 메소드를 사용할 것인지 선택하려면 두 가지 형식의 바인딩을 프로토타입해 보십시오. 정적 및 동적 SQL의 상세한 비교는 147 페이지의 『동적 SQL과 정적 SQL 비교』의 내용을 참조하십시오.

시간소인

패키지 또는 바인드 파일을 생성할 때 사전 처리 컴파일러는 시간소인을 생성합니다. 시간소인은 바인드 파일이나 패키지 및 수정된 소스 파일에 저장됩니다.

바인딩이 작동된 상태에서 응용프로그램이 사전 처리 컴파일될 때 패키지 및 수정된 소스 파일은 일치하는 시간 소인과 함께 생성됩니다. 응용프로그램이 수행될 때 시간소인이 같은지 점검됩니다. 응용프로그램 및 관련 패키지는 수행 응용프로그램의 시가소인이 일치해야 합니다. 그렇지 않으면 응용프로그램에 SQL0818N 오류가 리턴됩니다.

응용프로그램을 데이터베이스에 바인드할 때, *PREP 명령에서 PACKAGE USING 옵션을 사용하여 기본값을 바꾸지 않는 한* 응용프로그램 이름의 처음 8자는 패키지 이름입니다. 이는, 동일한 이름을 사용하여 두 프로그램을 사전 처리 컴파일 및 바인드할 경우 두 번째 프로그램이 첫번째 프로그램의 패키지를 겹쳐쓰게 됨을 의미합니다. 첫번째 프로그램을 수행할 때, 수정된 소스 파일에 대한 시간소인이 데이터베이스에 있는 패키지에서와 일치하지 않으므로 시간소인 오류가 발생하게 됩니다.

응용프로그램이 바인딩이 지연된 상태에서 사전 처리 컴파일될 때, 일치하는 시간소인을 가진 바인드 파일 및 수정된 소스 파일이 하나 이상 생성됩니다. 응용프로그램을 수행하기 위해 응용프로그램 모듈에 의해 작성된 바인드 파일을 실행할 수 있습니다. 바인딩 프로세스는 65 페이지의 『바인딩』에서 논의된 대로 각 바인드 파일에 대해 수행되어야 합니다.

사전 처리 컴파일 중에 수정된 소스 파일에 저장된 것과 바인드 파일의 시간소인이 같으므로, 응용프로그램 및 패키지 시간소인이 일치합니다.

리바인딩

*리바인드*란 이전에 바인드된 응용프로그램에 대한 패키지를 재작성하는 프로세스를 말합니다. 패키지가 올바르지 않은 또는 작동 불능으로 표시되었을 경우 패키지를 리바인드해야 합니다. 그러나, 올바른 패키지를 리바인드해야 할 경우도 있습니다. 예를 들어, RUNSTATS 명령을 수행한 후 갱신된 통계를 사용하거나 새로 작성된 색인을 이용하고자 할 수 있습니다.

패키지는 테이블, 뷰, 별명, 색인, 트리거, 참조 제한조건 및 테이블 점검 제한조건과 같은 일부 유형의 데이터베이스 오브젝트에 따라 달라질 수 있습니다. 패키지가 (테이블, 뷰, 트리거 등의) 데이터베이스 오브젝트에 종속적이고 그 오브젝트가 드롭되는 경우, 패키지는 *올바르지 않은* 상태로 됩니다. 드롭되는 오브젝트가 UDF인 경우 패키지는 *작동 불능* 상태로 됩니다. *관리 안내서: 계획*에서 자세한 정보를 참조하십시오

올바르지 않은 패키지는 실행될 때 데이터베이스 관리 프로그램에 의해 내재적으로 (또는 자동으로) 리바인드됩니다. 작동 불능 패키지는 BIND 명령이나 REBIND 명령을 실행하여 명시적으로 리바인드되어야 합니다. 내재된 리바인드를 통해 내재된 리바인드가 실패할 경우 예기치 못한 오류가 발생할 수 있습니다. 즉, 실제로 오류 상태인 명령문이 아닌 실행 중인 명령문에서 내재된 리바인드 오류가 리턴됩니다. 작동 불능 패키지를 실행하려 하면 오류가 발생합니다. 시스템에 의해 자동으로 리바인드되도록 하지 않고, 올바르지 않은 패키지를 명시적으로 리바인드할 수 있습니다. 이렇게 하면 리바인드할 때 제어를 할 수 있습니다.

패키지를 명시적으로 리바인드할 때 사용할 명령을 선택하는 것은 상황에 따라 다릅니다. BIND 명령을 사용하여 더 많거나 적은, 또는 변경된 SQL문을 포함하도

록 수정된 프로그램에 대한 패키지를 리바인드해야 합니다. 또한, 패키지가 원래 바인드되었던 값으로부터 바인드 옵션을 변경해야 할 경우 BIND 명령도 사용해야 합니다. 다른 경우, BIND 또는 REBIND 명령을 사용하십시오. 상황에 따라 BIND를 사용하지 않아도 되면 REBIND를 사용해야 합니다. REBIND가 BIND보다 성능이 월등히 낫기 때문입니다.

REBIND 명령에 대해서는 *Command Reference*의 내용을 참조하십시오.

제4장 정적 SQL 프로그램 작성

정적 SQL 사용 특성 및 이유 76
정적 SQL의 이점 76
예: 정적 SQL 프로그램 77
정적 프로그램 작업 방법. 78
C 예: STATIC.SQC 80
Java 예: Static.sqlj 81
COBOL 예: STATIC.SQB. 82
데이터 검색 및 조작용 SQL문 코딩 83
데이터 검색 83
호스트 변수 사용 84
선언 생성기 - db2dclgn 86
표시기 변수 사용 89
데이터 유형 91
STATIC 프로그램의 표시기 변수 사용 . . 94
커서를 이용한 다중 행 선택. 95
커서 사용 및 선언 96
커서 및 작업 단위(UOW) 고려사항 . . . 97
읽기 전용 커서 97
WITH HOLD 옵션 97
예: 커서 프로그램 99
커서 프로그램 작업 방법 100
C 예: CURSOR.SQC 101
Java 예: Cursor.sqlj. 103
COBOL 예: CURSOR.SQB 105
검색된 데이터 갱신 및 삭제 107
검색된 데이터 갱신 107
검색된 데이터 삭제 107
커서 유형 108
예: OPENFTCH 프로그램. 109
OPENFTCH 프로그램 작업 방법 . . 109
C 예: OPENFTCH.SQC 111
Java 예: Openftch.sqlj 113
COBOL 예: OPENFTCH.SQB . . . 116
고급 화면이동 기술 118
이미 검색된 데이터의 화면이동 118
데이터 사본 보관 118
데이터 두 번째 검색. 119
시작부터 검색 119
중간부터 검색 119
두 번째 결과 테이블의 행 순서 . . . 120
역순 검색 121
테이블 끝에 위치 설정 121
이전에 검색된 데이터 갱신. 121
예: UPDAT 프로그램 122
UPDAT 프로그램 작업 방법 122
C 예: UPDAT.SQC. 125
Java 예: Updat.sqlj 127
COBOL 예: UPDAT.SQB 129
REXX 예: UPDAT.CMD. 131
진단 조절 및 SQLCA 구조 133
리턴 코드 133
SQLCODE 및 SQLSTATE 133
SQLCA 구조에서의 토큰 절단 134
WHENEVER문 사용시 오류 처리 . . . 135
예외, 신호, 인터럽트 핸들러 고려사항 136
exit 목록 루틴 고려사항 137
예제 프로그램에서 GET ERROR MESSAGE 사용 137
C 예: UTILAPI.C 137
Java 예: SQLException 캐치 . . . 140
COBOL 예: CHECKERR.CBL. . . 141
REXX 예: CHECKERR 프로시듀어 143

© Copyright IBM Corp. 1993, 2000

정적 SQL 사용 특성 및 이유

Embedded SQL문이 사전 처리 컴파일시 완전히 알려진 경우 이러한 명령문을 *정적* SQL이라고 합니다. 이는 수행될 때까지 구문을 알 수 없는 *동적* SQL문과 대비됩니다.

주: 정적 SQL은 REXX와 같은 해석 언어에서는 지원되지 않습니다.

명령문이 정적으로 간주되기 위해서는 SQL문 *구조*가 완전히 지정되어야 합니다. 예를 들어, 명령문에서 참조되는 테이블 및 컬럼 이름은 사전 처리 컴파일될 때 완전히 알려져야 합니다. 런타임시에 지정될 수 있는 유일한 정보는 명령문에 의해 참조되는 호스트 변수에 대한 값입니다. 그러나 데이터 유형과 같은 호스트 변수 정보도 사전 처리 컴파일되어야 합니다.

정적 SQL문이 준비될 때 명령문의 실행 형식은 데이터베이스의 패키지에 작성 및 저장됩니다. 실행 형식은 사전 처리 컴파일될 때 또는 나중에 바인드될 때 구성될 수 있습니다. 어떤 경우이든 수행되기 *전*에 준비가 발생합니다. 응용프로그램 바인딩 담당자의 권한 부여가 사용되며, 최적화는 응용프로그램이 수행될 때 사용될 수 없는 구성 매개변수 및 데이터베이스 통계에 기초한 것입니다.

정적 SQL의 이점

정적 SQL을 사용하는 프로그래밍의 경우 embedded 동적 SQL을 사용할 때보다 노력이 절감됩니다. 정적 SQL문은 호스트 언어 소스 파일로만 포함되고, 사전 처리 컴파일러는 호스트 언어 컴파일러가 처리할 수 있는 데이터베이스 관리 프로그램 런타임 서비스 API 호출로의 필요한 변환을 처리합니다.

응용프로그램 바인딩 담당자의 권한 부여가 사용되므로 일반 사용자는 패키지 내의 명령문을 실행하기 위한 직접적인 특권이 필요하지 않습니다. 예를 들어, 응용프로그램은 전체 테이블에 대한 갱신 특권을 권한 부여하지 않고 사용자가 테이블 일부를 갱신할 수 있도록 할 수 있습니다. 이는 정적 SQL문 갱신을 특정 컬럼 또는 일정 범위의 값으로 제한함으로써 이루어집니다.

정적 SQL문은 *지속적*입니다. 즉, 명령문은 패키지가 존재하는 한 지속됩니다. 동적 SQL문은 무효화되거나 공간 관리상 비워질 때까지 또는 데이터베이스가 종료

될 때까지 캐쉬됩니다. 필요한 경우 동적 SQL문은 캐쉬된 명령문이 무효화될 때마다 DB2 SQL 컴파일러에 의해 내재적으로 재컴파일됩니다. 캐쉬된 명령문의 무효화 이유 및 캐쉬에 대해서는 *SQL 참조서*의 내용을 참조하십시오.

지속성과 관련하여 정적 SQL의 주요 이점은, 동적 SQL문은 특정 데이터베이스가 종료될 때 없어지는 반면 정적 SQL은 데이터베이스가 종료된 후에도 존재합니다. 또한, 정적 SQL은 런타임시에 DB2 SQL 컴파일러에 의해 컴파일될 필요가 없는 반면, 동적 SQL은 런타임시에 명시적으로(예를 들면, PREPARE문을 사용하여) 컴파일되어야 합니다. DB2는 동적 SQL문을 캐쉬하므로 DB2에 의해 명령문이 컴파일될 필요가 없지만, 응용프로그램을 실행할 때에는 적어도 한 번 컴파일되어야 합니다.

정적 SQL에는 성능상의 이점이 있을 수 있습니다. 간단한 단기 수행 SQL 프로그램의 경우, 정적 SQL문은 동적으로 처리되는 동일한 명령문보다 빨리 실행됩니다. 명령문의 실행 형식 준비 오버헤드가 수행될 때가 아닌 사전 처리 컴파일될 때 수행되기 때문입니다.

주: 정적 SQL의 성능은 응용프로그램은 마지막으로 응용프로그램이 바인딩되었을 때의 데이터베이스 통계에 따라 달라집니다. 그러나, 이러한 통계가 변경될 경우 해당 동적 SQL의 성능은 매우 달라질 수 있습니다. 예를 들어, 나중에 색인이 데이터베이스에 추가되면 정적 SQL을 사용하는 응용프로그램은 데이터베이스로 재바인드되지 않는 한 색인의 이점을 이용할 수 없습니다. 또한, 정적 SQL문에 호스트 변수를 사용할 경우, 최적화 알고리즘은 테이블에 대한 분배 통계를 이용할 수 없게 됩니다.

예: 정적 SQL 프로그램

이 샘플 프로그램은 다음과 같은 지원 언어로 정적 SQL문 및 데이터베이스 관리 프로그램 API 호출의 예를 보여 줍니다.

C	static.sqc
Java	Static.sqlj
COBOL	static.sqb

REXX 언어는 정적 SQL을 지원하지 않으므로 샘플은 제공되지 않습니다.

이 샘플 프로그램에는 단일 행을 선택하는 조회가 포함되어 있습니다. 이러한 조회는 SELECT INTO문을 사용하여 수행할 수 있습니다.

SELECT INTO문은 데이터베이스의 테이블에서 한 행의 데이터를 선택하고, 이 행의 값들은 명령문에 지정된 호스트 변수에 지정됩니다. 호스트 변수에 대해서는 84 페이지의 『호스트 변수 사용』에 상세히 논의되어 있습니다. 예를 들어, 다음 명령문은 성이 'HAAS'인 사원의 급여를 empsal 호스트 변수로 전달합니다.

```
    SELECT SALARY
      INTO :empsal
 FROM EMPLOYEE
      WHERE LASTNAME='HAAS'
```

SELECT INTO문은 한 행만 리턴하도록 지정해야 합니다. 한 행 이상을 발견하면 오류, SQLCODE -811(SQLSTATE 21000)이 발생합니다. 조회 결과 여러 행이 나올 경우 커서를 사용하여 행을 처리해야 합니다. 95 페이지의 『커서를 이용한 다중 행 선택』에서 자세한 내용을 참조하십시오.

SELECT INTO문에 대한 세부사항은 *SQL 참조서*의 내용을 참조하십시오.

SELECT문 작성 방법에 대한 소개는 83 페이지의 『데이터 검색 및 조작용 SQL문 코딩』의 내용을 참조하십시오.

정적 프로그램 작업 방법

1. **SQLCA 포함.** INCLUDE SQLCA문은 SQLCA 구조를 정의 및 선언하고, 구조 내의 요소로서 SQLCODE 및 SQLSTATE를 정의합니다. SQLCA 구조의 SQLCODE 필드는 SQL문이나 데이터베이스 관리 프로그램 API 호출이 실행될 때마다 데이터베이스 관리 프로그램에 의해 진단 정보로 갱신됩니다.
2. **호스트 변수 정의.** SQL BEGIN DECLARE SECTION문 및 END DECLARE SECTION문은 호스트 변수 선언을 분리합니다. 이는 SQL문에서 참조될 수 있는 변수들입니다. 호스트 변수를 사용하여 데이터를 데이터베이스 관리 프로그램으로 전달하거나 리턴되는 데이터를 보유합니다. 호스트 변수는 SQL문에서 참조될 때 콜론(:)으로 시작됩니다. 84 페이지의 『호스트 변수 사용』에서 자세한 정보를 참조하십시오.

3. **데이터베이스에 연결.** 프로그램은 sample 데이터베이스에 연결하여 이 데이터베이스에 대한 공유 액세스를 요청합니다.(START DATABASE MANAGER API 호출이나 db2start 명령이 발행되었다고 전제합니다.) 공유 액세스를 사용하여 동일한 데이터베이스에 연결하는 다른 프로그램들에게도 액세스가 권한 부여됩니다.
4. **데이터 검색.** SELECT INTO문은 조회를 기초로 하여 단일 값을 검색합니다. 이 예에서는 LASTNAME 컬럼 값이 JOHNSON인 EMPLOYEE 테이블로부터 FIRSTNME 컬럼을 검색합니다. SYBIL 값이 리턴되어 firstname 호스트 변수에 놓입니다. DBw에서 제공되는 샘플 테이블은 *SQL 참조서*의 부록에 나열됩니다.
5. **오류 처리.** CHECKERR 매크로/함수는 오류 체크 유틸리티입니다. 이 오류 체크 유틸리티의 위치는 사용된 프로그래밍 언어에 따라 달라질 수 있습니다.

   C DB2 API를 호출하는 C 프로그램에 대해서는 utilapi.c에서 sqlInfoPrint 함수는 utilapi.h에서 API_SQL_CHECK로서 참조됩니다. C embedded SQL 프로그램에 대해서는 utilemb.sqc에서 sqlInfoPrint 함수는 utilemb.h에서 EMB_SQL_CHECK로서 참조됩니다.

   Java SQL 오류는 SQLException으로서 드로우되고, 응용프로그램의 catch 블록에서 처리됩니다.

   COBOL CHECKERR은 checkerr.cbl라고 하는 외부 프로그램입니다.

   이 오류 체크 유틸리티에 대한 소스 코드에 대해서는 137 페이지의『예제 프로그램에서 GET ERROR MESSAGE 사용』의 내용을 참조하십시오.
6. **데이터베이스에서 연결해제.** 프로그램은 CONNECT RESET문을 실행하여 데이터베이스로부터의 연결을 해제합니다. SQLJ 프로그램은 프로그램이 리턴될 때 데이터베이스 연결을 자동으로 닫습니다.

C 예: STATIC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;   1

int main(int argc, char *argv[])
{   int  rc = 0;

    char dbAlias[15] ;
    char user[15] ;
    char pswd[15] ;

    EXEC SQL BEGIN DECLARE SECTION;  2
        char firstname[13];
    EXEC SQL END DECLARE SECTION;

    /* checks the command line arguments */
    rc = CmdLineArgsCheck1( argc, argv, dbAlias, user, pswd );  3
    if ( rc != 0 ) return( rc ) ;

    printf("\n\nSample C program: STATIC\n");

    /* initialize the embedded application */
    rc = EmbAppInit( dbAlias, user, pswd);
    if ( rc != 0 ) return( rc ) ;

    EXEC SQL SELECT FIRSTNME INTO :firstname   4
            FROM employee
            WHERE LASTNAME = 'JOHNSON';
    EMB_SQL_CHECK("SELECT statement");   5

    printf( "First name = %s\n", firstname );

    /* terminate the embedded application */
    rc = EmbAppTerm( dbAlias);
    return( rc ) ;
}
/* end of program : STATIC.SQC */
```

Java 예: Static.sqlj

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

class Static
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
      }
      catch (Exception e)
      {   System.out.println ("\n  Error loading DB2 Driver...\n");
        System.out.println (e);
        System.exit(1);
      }
  }

  public static void main(String argv[])
  {   try
    {   System.out.println ("  Java Static Sample");

      String url = "jdbc:db2:sample";       // URL is jdbc:db2:dbname
      Connection con = null;

      // Set the connection                   3
      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception("\nUsage: java Static [username password]\n");
      }

      // Set the default context
      DefaultContext ctx = new DefaultContext(con);
      DefaultContext.setDefaultContext(ctx);

      String firstname = null;

      #sql { SELECT FIRSTNME INTO :firstname
           FROM employee
           WHERE LASTNAME = 'JOHNSON' } ;    4


      System.out.println ("First name = " + firstname);
    }

    catch( Exception e )  5
    {
      System.out.println (e);
    }
  }
}
```

COBOL 예: STATIC.SQB

```
Identification Division.
Program-ID. "static".

Data Division.
Working-Storage Section.

    copy "sql.cbl".
    copy "sqlca.cbl".                 1

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.    2
01 firstname         pic x(12).
01 userid            pic x(8).
01 passwd.
  49 passwd-length   pic s9(4) comp-5 value 0.
  49 passwd-name     pic x(18).
    EXEC SQL END DECLARE SECTION END-EXEC.

77 errloc          pic x(80).

Procedure Division.
Main Section.
    display "Sample COBOL program: STATIC".

    display "Enter your user id (default none): "
         with no advancing.
    accept userid.

    if userid = spaces
      EXEC SQL CONNECT TO sample END-EXEC
    else
      display "Enter your password : " with no advancing
      accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.
    inspect passwd-name tallying passwd-length for characters
       before initial " ".

    EXEC SQL CONNECT TO sample USER :userid USING :passwd    3
       END-EXEC.
    move "CONNECT TO" to errloc.
    call "checkerr" using SQLCA errloc.

    EXEC SQL SELECT FIRSTNME INTO :firstname         4
             FROM EMPLOYEE
             WHERE LASTNAME = 'JOHNSON' END-EXEC.
    move "SELECT" to errloc.
    call "checkerr" using SQLCA errloc.          5

    display "First name = ", firstname.

    EXEC SQL CONNECT RESET END-EXEC.            6
    move "CONNECT RESET" to errloc.
    call "checkerr" using SQLCA errloc.

End-Prog.
    stop run.
```

데이터 검색 및 조작용 SQL문 코딩

데이터베이스 관리 프로그램은 응용프로그램 프로그래머에게 데이터 검색 및 조작용 명령문을 제공합니다. 코딩 타스크는 이러한 명령문을 호스트 언어 코드에 포함시키는 것으로 구성됩니다. 이 절에서는 DB2 테이블에 있는 한 행 이상의 데이터에서 데이터를 검색 및 조작할 명령문을 코드화하는 방법에 대해 설명합니다.(다른 호스트 언어의 세부사항으로 가지 않습니다.) SQL문의 위치 지정, 지속 및 분리 규칙에 대해서는 다음을 참조하십시오.

- 661 페이지의 『제20장 C 및 C++ 프로그래밍』
- 709 페이지의 『제21장 Java 프로그래밍』
- 761 페이지의 『제23장 COBOL 프로그래밍』
- 787 페이지의 『제24장 FORTRAN 프로그래밍』
- 805 페이지의 『제25장 REXX 프로그래밍』.

데이터 검색

SQL 응용프로그램에서 가장 공통된 타스크는 데이터를 검색하는 것입니다. 이는 지정된 검색 조건을 충족시키는 데이터베이스에 있는 테이블 행을 검색하는 조회 형식인 *select-statement*를 사용하여 수행됩니다. 이러한 행이 존재하는 경우, 데이터가 검색되어 호스트 프로그램의 지정된 변수에 놓이게 됩니다. 호스트 프로그램에서 이 데이터는 설계 목적에 따라 사용될 수 있습니다.

*select-statement*를 작성한 후에는 정보가 응용프로그램으로 전달되는 방식을 정의하는 SQL문을 코드화할 수 있습니다.

select-statement 결과는 데이터베이스에 있는 테이블과 마찬가지로, 행 및 컬럼이 있는 테이블로 생각할 수 있습니다. 한 행만 리턴되는 경우 SELECT INTO문에 의해 지정되는 호스트 변수로 직접 결과를 전달할 수 있습니다.

한 행 이상이 리턴되는 경우 *커서*를 사용하여 한 번에 하나를 페치해야 합니다. 커서는 응용프로그램에 의해 사용되는 명명된 제어 구조로서, 순서화된 행 세트 내의 특정 행을 가리킵니다. 커서의 코드화 및 사용 방법에 대해서는 다음 절을 참조하십시오.

- 96 페이지의 『커서 사용 및 선언』,
- 95 페이지의 『커서를 이용한 다중 행 선택』,

- 99 페이지의『예: 커서 프로그램』.

호스트 변수 사용

*호스트 변수*는 Embedded SQL문에 의해 참조되는 변수입니다. 이들은 데이터베이스 관리 프로그램과 응용프로그램 사이의 데이터를 전송합니다. *SQL문*에서 호스트 변수를 사용할 때 콜론과 함께 이름으로 접두부를 구성해야 합니다(예. **:**). *호스트 언어 명령문*에서 호스트 변수를 사용할 때에는 콜론을 생략하십시오.

호스트 변수는 컴파일된 호스트 언어로 선언되고, BEGIN DECLARE SECTION 문 및 END DECLARE SECTION문으로 분리됩니다. 이러한 명령문을 이용하여 사전 처리 컴파일러가 선언문을 발견할 수 있습니다.

주: Java JDBC 및 SQLJ 프로그램은 선언 절을 사용하지 않습니다. Java어로 된 호스트 변수는 일반적인 Java 변수 선언 구문을 따릅니다.

호스트 변수는 호스트 언어의 부속 집합을 사용하여 선언됩니다. 사용자의 호스트 언어에 대한 지원 구문에 대한 설명은 다음을 참조하십시오.

- 661 페이지의『제20장 C 및 C++ 프로그래밍』
- 709 페이지의『제21장 Java 프로그래밍』
- 761 페이지의『제23장 COBOL 프로그래밍』
- 787 페이지의『제24장 FORTRAN 프로그래밍』
- 805 페이지의『제25장 REXX 프로그래밍』.

다음 규칙은 호스트 변수 선언 절에 적용됩니다.

- 모든 호스트 변수는 SQLDA 구조를 참조하는 호스트 변수를 제외하고, 참조되기 전에 소스 파일에서 선언되어야 합니다.
- 한 소스 파일에서 다중 선언 절을 사용할 수 있습니다.
- 사전 처리 컴파일러는 호스트 언어 변수 범위설정 규칙을 인식하지 못합니다.

  SQL문을 볼 때, 단일 소스 파일에서 실제로 선언되는 위치와 관계없이 모든 호스트 변수에는 전체 범위를 가집니다. 따라서, 호스트 변수 이름은 소스 파일 내에서 고유해야 합니다.

이는 DB2 사전 처리 컴파일러가 호스트 변수의 범위를 전역으로 변경하여, 정의되는 범위 외부에서 액세스될 수 있음을 의미하지는 않습니다. 다음 예를 고려하십시오.

```
  foo1(){
  .
  .
  .
    BEGIN SQL DECLARE SECTION;
    int x;
    END SQL DECLARE SECTION;
    x=10;
  .
  .
  .
}


foo2(){
  .
  .
  .
    y=x;
  .
  .
  .
}
```

언어에 따라 x 변수는 foo2() 함수에서 선언되지 않거나 x 값이 foo2()로 10으로 설정되지 않으므로 위 예는 컴파일하는 데 실패합니다. 이러한 문제를 피하려면 전역 변수로서 x를 선언하거나, 매개변수로서 x를 다음과 같이 foo2() 함수로 전달해야 합니다.

```
  foo1(){
  .
  .
  .
    BEGIN SQL DECLARE SECTION;
    int x;
    END SQL DECLARE SECTION;
    x=10;
    foo2(x);
  .
  .
  .
  }
```

```
foo2(int x){
.
.
.
  y=x;
.
.
.
}
```

호스트 변수 선언에 대한 자세한 정보는 다음을 참조하십시오.

- db2dclgn 도구로 자동으로 호스트 변수 선언 소스 코드를 생성하려면 『선언 생성기 - db2dclgn』의 내용을 참조하십시오.
- 호스트 변수가 소스 코드로 나타나는 방식의 예는 87 페이지의 표4의 내용을 참조하십시오.
- 지원되는 호스트 언어로 호스트 변수를 참조하는 방식의 예는 88 페이지의 표5의 내용을 참조하십시오.
- REXX로 호스트 변수를 명명 및 참조하는 것에 대해서는 810 페이지의 『REXX에서 호스트 변수 명명』 및 810 페이지의 『REXX에서 호스트 변수 조회』의 내용을 참조하십시오.

선언 생성기 - db2dclgn

Declaration Generator는 데이터베이스에 제공된 테이블에 대해 선언문을 생성하여 응용프로그램 개발을 가속화합니다. 이 유틸리티는 사용자 응용프로그램에 쉽게 삽입할 수 있는 Embedded SQL 선언문 소스 파일을 작성합니다. db2dclgn은 C/C++, Java, COBOL 및 FORTRAN 언어를 지원합니다.

선언 파일을 생성하려면 다음 형식으로 db2dclgn 명령을 입력하십시오.

```
db2dclgn -d database-name -t table-name
[options]
```

예를 들어, staff.h 출력 파일에서 C로 된 SAMPLE 데이터베이스에 STAFF 테이블에 대한 선언문을 생성하려면 다음 명령을 발행하십시오.

```
db2dclgn -d sample -t staff -l C
```

결과 staff.h 파일에는 다음이 포함됩니다.

```
struct
{
  short id;
  struct
  {
    short length;
    char data[9];
  } name;
  short dept;
  char job[5];
  short years;
  double salary;
  double comm;
} staff;
```

db2dclgn에서 자세한 정보는 *Command Reference*를 참조하십시오.

표 4. 호스트 변수 선언

언어	예제 소스 코드

C/C++	`EXEC SQL BEGIN DECLARE SECTION;`
	`  short     dept=38, age=26;`
	`  double    salary;`
	`  char      CH;`
	`  char      name1[9], NAME2[9];`
	`  /* C comment */`
	`  short     nul_ind;`
	`EXEC SQL END DECLARE SECTION;`

Java	`// Note that Java host variable declarations follow`
	`// normal Java variable declaration rules, and have`
	`// no equivalent of a DECLARE SECTION`
	`  short     dept=38, age=26;`
	`  double    salary;`
	`  char      CH;`
	`  String    name1[9], NAME2[9];`
	`  /* Java comment */`
	`  short     nul_ind;`

표 4. 호스트 변수 선언 (계속)

<table>
<tr><th>언어</th><th>예제 소스 코드</th></tr>
<tr><td>COBOL</td><td><pre>
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   01 age   PIC S9(4) COMP-5 VALUE 26.
   01 DEPT   PIC S9(9) COMP-5 VALUE 38.
   01 salary   PIC S9(6)V9(3) COMP-3.
   01 CH   PIC X(1).
   01 name1   PIC X(8).
   01 NAME2   PIC X(8).
* COBOL comment
   01 nul-ind   PIC S9(4) COMP-5.
 EXEC SQL END DECLARE SECTION END-EXEC.
</pre></td></tr>
<tr><td>FORTRAN</td><td><pre>
       EXEC SQL BEGIN DECLARE SECTION
         integer*2   age   /26/
         integer*4   dept /38/
         real*8   salary
         character   ch
         character*8   name1,NAME2
C   FORTRAN comment
         integer*2   nul_ind
       EXEC SQL END DECLARE SECTION
</pre></td></tr>
</table>

표 5. 호스트 변수 참조

<table>
<tr><th>언어</th><th>예제 소스 코드</th></tr>
<tr><td>C/C++</td><td><pre>
EXEC SQL FETCH C1 INTO :cm;
printf("Commission = %f\n", cm);
</pre></td></tr>
<tr><td>JAVA (SQLJ)</td><td><pre>
#SQL { FETCH :c1 INTO :cm };
System.out.println("Commission = " + cm);
</pre></td></tr>
<tr><td>COBOL</td><td><pre>
EXEC SQL FETCH C1 INTO :cm END-EXEC
DISPLAY 'Commission = ' cm
</pre></td></tr>
<tr><td>FORTRAN</td><td><pre>
EXEC SQL FETCH C1 INTO :cm
WRITE(*,*) 'Commission = ', cm
</pre></td></tr>
</table>

표시기 변수 사용

Java가 아닌 언어로 작성된 응용프로그램은 널(NULL) 값을 받을 수 있는 호스트 변수와 *표시기 변수*를 연결하여 널(NULL) 값 수신을 준비해야 합니다. Java 응용프로그램은 호스트 변수의 값을 Java 널(NULL)을 비교하여 수신된 값이 널(NULL)인지 결정합니다. 표시기 변수는 데이터베이스 관리 프로그램 및 호스트 응용프로그램 모두가 공유하므로, 표시기 변수는 호스트 변수로서 응용프로그램에서 선언되어야 합니다. 이 호스트 변수는 SQL 데이터 유형 SMALLINT에 일치합니다.

표시기 변수는 SQL문에서 호스트 변수 바로 뒤에 오고, 접두부가 콜론입니다. 공간은 호스트 변수로부터 표시기 변수를 분리할 수 있지만 필요하지는 않습니다. 그러나, 호스트 변수와 표시기 변수 사이에 쉼표를 넣지 마십시오. 또한, 호스트 변수와 표시기 사이에 선택적인 INDICATOR 키워드를 사용하여 표시기 변수를 지정할 수도 있습니다.

표시기 변수에는 INDICATOR 키워드를 사용하여 지원 호스트 언어로 표시기 변수 용례가 나와 있습니다.

언어 **예제 소스 코드**

C/C++

```
EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind;
if ( cmind < 0 )
   printf( "Commission is NULL\n" );
```

Java (SQLJ)

```
#SQL { FETCH :c1 INTO :cm };
if ( cm == null )
   System.out.println( "Commission is NULL\n" );
```

COBOL

```
EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind END-EXEC
IF cmind LESS THAN 0
   DISPLAY 'Commission is NULL'
```

FORTRAN

```
EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind
IF ( cmind .LT. 0 ) THEN
   WRITE(*,*) 'Commission is NULL'
ENDIF
```

그림에서 *cmind*가 음수값인지 검토됩니다. 음수가 아닌 경우 응용프로그램은 *cm* 의 리턴 값을 사용할 수 있습니다. 음수인 경우 페치된 값은 널(NULL)이고 *cm* 은 사용할 수 없습니다. 데이터베이스 관리 프로그램은 이런 경우 호스트 변수 값을 변경하지 않습니다.

주: 데이터베이스 구성 매개변수 DFT_SQLMATHWARN이 'YES'로 설정되면 *cmind* 값은 -2입니다. 이는 수치 결과 값을 호스트 변수로 변환하는 중에 발생한 오버플로우 또는 산술 오류가 있는 표현식을 산정한 결과 발생한 널(NULL)임을 나타냅니다.

데이터 유형이 널(NULL)을 처리할 수 있는 경우 응용프로그램은 NULL 표시기를 제공해야 합니다. 그렇지 않으면 오류가 발생할 수 있습니다. NULL 표시기가 사용되지 않으면 SQLCODE -305(SQLSTATE 22002)가 리턴됩니다.

SQLCA 구조가 절단 경고를 표시할 경우 표시기 변수에 절단이 있는지 검토됩니다. 표시기 변수가 양수이면 절단이 발생한 것입니다.

- TIME 데이터 유형의 초 부분이 절단될 경우 표시기 값에는 절단된 데이터의 초 부분이 포함됩니다.
- 대형 오브젝트(LOB)를 제외한 다른 모든 문자열 데이터 유형의 경우 표시기 값은 리턴된 데이터의 실제 길이를 나타냅니다. 사용자 정의 구별 유형(UDT)은 기본 유형과 같은 방식으로 처리됩니다.

INSERT문 또는 UPDATE문을 처리할 때 데이터베이스 관리 프로그램은 표시기 변수를 점검합니다. 표시기 변수가 음수이면, 데이터베이스 관리 프로그램은 NULL 이 허용되는 경우 목표 컬럼 값을 NULL로 설정합니다. 표시기 변수가 제로 또는 양수이면 데이터베이스 관리 프로그램은 연관된 호스트 변수 값을 사용합니다.

호스트 변수에 지정될 때 문자열 컬럼 값이 절단될 경우 SQLCA 구조의 SQLWARN1 필드에 'X' 또는 'W'가 포함될 수 있습니다. 널(NULL) 종료기가 절단될 경우 'N'이 포함됩니다.

’X’ 값은 다음 모든 조건이 충족되는 경우에만 데이터베이스 관리 프로그램에 의해 리턴됩니다.

- 데이터베이스 코드 페이지로부터 응용프로그램 코드 페이지로의 문자열 데이터 변환에 데이터 길이 변경이 포함되는 혼합 코드 페이지 연결이 존재합니다.
- 커서가 블록되어 있습니다.
- 사용자 응용프로그램에 의해 표시기 변수가 제공됩니다.

표시기 변수에서 리턴되는 값은 응용프로그램의 코드 페이지에 있는 결과 문자열의 길이가 됩니다.

데이터 절단이 포함된 기타 모든 경우 (NULL 종료기 절단과 반대로) 데이터베이스 관리 프로그램은 ’W’를 리턴합니다. 이런 경우 데이터베이스 관리 프로그램은, 선택 목록 항목(응용프로그램 코드 페이지나 데이터 기본 코드 페이지 또는 두 항목 모두 아님)의 코드 페이지에서 결과 문자열 길이인 표시기 변수의 값을 응용프로그램에 리턴합니다. 관련 정보는 *SQL 참조서*의 내용을 참조하십시오.

데이터 유형

컬럼이 작성될 때 모든 DB2 테이블의 각 컬럼에는 *SQL 데이터 유형*이 제공됩니다. 이러한 유형이 컬럼에 지정되는 방식에 대해서는 *SQL 참조서*의 CREATE TABLE문을 참조하십시오. 데이터베이스 관리 프로그램은 다음과 같은 컬럼 데이터 유형을 지원합니다.

SMALLINT

16 비트 부호있는 정수

INTEGER

32 비트 부호있는 정수 **INT**는 이 유형의 동의어로 사용할 수 있습니다.

BIGINT

64 비트 부호있는 정수

DOUBLE

배정밀 부동 소수점. **DOUBLE PRECISION** 및 **FLOAT(n)**(n이 24보다 큰 경우)는 이 유형의 동의어입니다.

REAL
: 단정밀 부동 소수점. **FLOAT(n)**(n이 24보다 작은 경우)는 이 유형의 동의어입니다.

DECIMAL
: 압축 십진수 **DEC**, **NUMERIC** 및 **NUM**은 이 유형의 동의어입니다.

CHAR
: 1바이트에서 254 바이트 길이인 고정 길이 문자열. **CHARACTER**는 이 유형의 동의어로 사용할 수 있습니다.

VARCHAR
: 1바이트에서 32672 바이트 길이인 가변 길이 문자열 **CHARACTER VARYING** 및 **CHAR VARYING**은 이 유형의 동의어입니다.

LONG VARCHAR
: 1바이트에서 32 700 바이트인 long 가변 길이 문자열.

CLOB
: 1바이트에서 2 기가바이트인 대형 오브젝트(LOB) 가변 길이 문자열.

BLOB
: 1바이트에서 2기가바이트인 대형 오브젝트(LOB) 가변 길이 2진 문자열.

DATE
: 날짜를 나타내는 길이가 10인 문자열.

TIME 시간을 나타내는 길이가 8인 문자열.

TIMESTAMP
: 시간소인을 나타내는 길이가 26인 문자열.

다음 데이터 유형은 2바이트 문자 세트(DBCS) 및 Extended UNIX Code(EUC) 문자 세트 환경에서만 지원됩니다.

GRAPHIC
: 1자-127자 길이의 2바이트 문자열의 고정 길이 그래픽 문자열.

VARGRAPHIC
: 1자-16336자의 2바이트 문자열의 가변 길이 그래픽 문자열.

LONG VARGRAPHIC

1자-16 350자 길이의 2바이트 문자열의 long 가변 길이 그래픽 문자열.

DBCLOB

길이가 1에서 1 073 741 823의 2바이트 문자열의 대형 오브젝트(LOB) 가변 길이 그래픽 문자열.

주:

1. 모든 지원 데이터 유형에는 NOT NULL 속성이 있을 수 있습니다. 이는 다른 유형으로서 취급됩니다.
2. 위 데이터 유형 세트는 사용자 정의 구별 유형(UDT)을 정의하여 확장될 수 있습니다. UDT는 내장된 SQL 유형 표시를 사용하는 구분된 데이터 유형입니다.

지원되는 호스트 언어에는 대부분의 데이터베이스 관리 프로그램 데이터 유형에 일치하는 데이터 유형이 있습니다. 호스트 변수 선언에서 이 호스트 언어 데이터 유형만 사용할 수 있습니다. 사전 처리 컴파일러기 호스트 변수 선언을 발견할 때 적절한 SQL 데이터 유형 값을 결정합니다. 데이터베이스 관리 프로그램은 이 값을 사용하여 자신과 응용프로그램 사이에 교환되는 데이터를 변환합니다.

응용프로그램 프로그래머로서 사용자가 데이터베이스 관리 프로그램이 다른 데이터 유형들 사이에서 비교 및 지정을 처리하는 방식을 이해하는 것이 중요합니다. 간단히 말해서 데이터 유형은 데이터베이스 관리 프로그램이 두 개의 SQL 컬럼 데이터 유형, 두 개의 호스트 언어 데이터 유형 또는 각 유형에 대해 작업하는지에 관계없이, 지정 및 비교 조작을 하는 중에 서로 호환되어야 합니다.

데이터 유형 호환성의 *일반* 규칙은 지원되는 모든 호스트 언어 수치 데이터 유형은 모든 데이터베이스 관리 프로그램 수치 데이터 유형으로 비교 및 지정 가능하며, 모든 호스트 언어 문자 유형은 모든 데이터베이스 관리 프로그램 문자 유형과 호환됩니다. 수치 유형은 문자 유형과 호환되지 않습니다. 그러나, 대형 오브젝트(LOB)에 대해 작업할 때 부과된 호스트 언어 특징 및 제한사항에 따라 이러한 일반 규칙에 대한 예외도 일부 있습니다.

SQL문 내에서 DB2는 호환가능한 데이터 유형 사이의 변환을 제공합니다. 예를 들어, 다음 SELECT문에서 SALARY 및 BONUS는 DECIMAL 컬럼입니다. 그러나 사원의 전체 보상은 DOUBLE 데이터로서 리턴됩니다.

```
SELECT EMPNO, DOUBLE(SALARY+BONUS) FROM EMPLOYEE
```

위 명령문의 실행에는 DECIMAL 및 DOUBLE 데이터 유형 사이의 변환이 포함됩니다. 조회 결과를 사용자 화면에서 보다 쉽게 읽을 수 있도록 하기 위해 다음 SELECT문을 사용할 수 있습니다.

```
SELECT EMPNO, DIGIT(SALARY+BONUS) FROM EMPLOYEE
```

응용프로그램 내에서 데이터를 변환하려면 추가 루틴, 클래스, 내장된 유형 또는 이 변환을 지원하는 API에 대한 사용자 컴파일러 제조업체에 문의해 보십시오.

문자 데이터 유형도 문자 변환이 될 수 있습니다. 사용자 응용프로그램 코드 페이지가 데이터베이스 코드 페이지와 같지 않은 경우 573 페이지의『다른 코드 페이지 간의 변환』의 내용을 참조하십시오.

지원 SQL 데이터 유형 및 해당 호스트 언어 데이터 유형 목록에 대해서는 다음을 참조하십시오.
- C/C++용, 701 페이지의『C 및 C++의 지원되는 SQL 데이터 유형』
- Java용, 712 페이지의『Java에서 지원되는 SQL 데이터 유형』
- COBOL용, 780 페이지의『COBOL에서 지원되는 SQL 데이터 유형』
- FORTRAN용, 800 페이지의『FORTRAN에서 지원되는 SQL 데이터 유형』
- REXX용, 817 페이지의『REXX에서 지원되는 SQL 데이터 유형』.

SQL 데이터 유형, 지정 및 비교 규칙, 데이터 변환 및 변환 오류에 대한 세부사항은 *SQL 참조서*의 내용을 참조하십시오.

STATIC 프로그램의 표시기 변수 사용

다음 코드 세그먼트는 80 페이지의『C 예: STATIC.SQC』에 나열된 샘플 STATIC 프로그램의 C 버전에서 해당 세그먼트에 대한 수정을 보여줍니다. 이들 세그먼트는 널(NULL) 입력 가능한 데이터 컬럼에서의 표시기 변수 구현을 나타냅니다. 예

에서, STATIC 프로그램이 다른 컬럼, WORKDEPT를 선택하도록 확장됩니다. 이 컬럼은 널(NULL) 값을 가질 수 있습니다. 표시기 변수는 사용되기 전에 호스트 변수로서 선언되어야 합니다.

```
⋮
EXEC SQL BEGIN DECLARE SECTION;
  char  wd[3];
  short wd_ind;
  char firstname[13];
⋮
EXEC SQL END DECLARE SECTION;
⋮
/* CONNECT TO SAMPLE DATABASE */
⋮
EXEC SQL SELECT FIRSTNME, WORKDEPT INTO :firstname, :wd:wdind
  FROM EMPLOYEE
  WHERE LASTNAME = 'JOHNSON';
⋮
```

커서를 이용한 다중 행 선택

응용프로그램이 행 세트를 검색하도록 하기 위해 SQL은 *커서*라는 메카니즘을 사용합니다.

커서라는 개념의 이해를 돕기 위해, 데이터베이스 관리 프로그램은 SELECT문을 실행하여 검색되는 모든 행들을 보유할 *결과 테이블*을 구축한다고 가정합니다. 커서는 이 테이블의 *현재 행*을 가리키거나 식별함으로써 응용프로그램이 사용할 수 있는 결과 테이블로부터 행을 작성합니다. 커서가 사용될 때 응용프로그램은 데이터 조건의 끝, 즉 NOT FOUND 조건, SQLCODE +100(SQLSTATE 02000)에 도달할 때까지 결과 테이블로부터 순차적으로 각 행을 검색할 수 있습니다. SELECT문 실행 결과 획득된 행 세트들은 검색 조건을 충족시키는 행 수에 따라 0개, 1개 또는 그 이상의 행들로 구성될 수 있습니다.

커서 처리 단계는 다음과 같습니다.

1. DECLARE CURSOR문을 사용하여 커서를 지정합니다.
2. 조회를 수행하고 OPEN문을 사용한 결과 테이블을 구축합니다.
3. FETCH문을 사용하여 한 번에 한 행을 검색합니다.
4. (필요한 경우) DELETE문 또는 UPDATE문이 있는 행을 처리합니다.
5. CLOSE문을 사용하여 커서를 종료합니다.

응용프로그램은 동시에 여러 커서를 사용할 수 있습니다. 커서마다 고유의 DECLARE CURSOR, OPEN, CLOSE 및 FETCH문 세트가 필요합니다.

응용프로그램이 행 세트를 선택하고, 커서를 사용하여 한 번에 한 행 설정을 처리하는 방법의 예는 99 페이지의『예: 커서 프로그램』의 내용을 참조하십시오.

커서 사용 및 선언

DECLARE CURSOR문은 SELECT문을 사용하여 검색할 행 세트를 식별하여 커서를 정의 및 명명합니다.

응용프로그램은 커서에 이름을 지정합니다. 이 이름은 후속 OPEN문, FETCH문 및 CLOSE문에서 참조됩니다. 조회는 유효한 선택 명령문입니다.

커서문 정의에는 정적 SELECT문과 연관된 DECLARE문이 있습니다.

언어 예제 소스 코드

C/C++

```
EXEC SQL DECLARE C1 CURSOR FOR
 SELECT PNAME, DEPT FROM STAFF
 WHERE JOB=:host_var;
```

Java (SQLJ)

```
#sql iterator cursor1(host_var data type);
#sql cursor1 = { SELECT PNAME, DEPT FROM STAFF
                 WHERE JOB=:host_var };
```

COBOL

```
EXEC SQL DECLARE C1 CURSOR FOR
 SELECT NAME, DEPT FROM STAFF
   WHERE JOB=:host-var END-EXEC.
```

FORTRAN

```
   EXEC SQL DECLARE C1 CURSOR FOR
  +  SELECT NAME, DEPT FROM STAFF
  +  WHERE JOB=:host_var
```

주: DECLARE문의 위치는 임의적이지만, 최초의 커서 사용 위에 위치해야 합니다.

커서 및 작업 단위(UOW) 고려사항

COMMIT 또는 ROLLBACK 조작의 조치는 커서가 선언되는 방식에 따라 커서마다 다릅니다.

읽기 전용 커서

커서가 읽기 전용이고 반복 읽기(RR) 분리 레벨을 사용하는 경우, 작업 단위(UOW)에 필요한 시스템 테이블에서 반복 읽기 잠금이 여전히 수집 및 유지보수됩니다. 따라서 응용프로그램은 읽기 전용 커서에 대해서도 COMMIT문을 정기적으로 발행해야 합니다.

WITH HOLD 옵션

응용프로그램이 COMMIT문을 발행하여 작업 단위(UOW)를 완료하는 경우, WITH HOLD 옵션 사용을 제외하고 모든 *개방 커서*는 데이터베이스 관리 프로그램에 의해 자동으로 닫힙니다.

WITH HOLD로 선언된 커서는 여러 개의 작업 단위(UOW)를 통해 액세스하는 자원을 유지보수합니다. 커서를 WITH HOLD로 선언하는 결과는 작업 단위(UOW)가 종료되는 방식에 따라 달라집니다.

작업 단위(UOW)가 COMMIT문으로 종료될 경우 WITH HOLD로 정의된 개방 커서는 OPEN 상태로 남아 있습니다. 커서는 결과 테이블의 다음 논리 행 앞에 위치합니다. 또한, WITH HOLD로 정의된 OPEN 커서를 참조하는 준비된 명령문이 보유됩니다. 특정 커서로 연관된 FETCH 및 CLOSE 요청만이 COMMIT 직후에 유효합니다. UPDATE WHERE CURRENT OF문 및 DELETE WHERE CURRENT OF문은 동일한 작업 단위(UOW) 내에서 페치된 행들에 대해서만 유효합니다. 패키지가 작업 단위(UOW) 중에 리바인드되는 경우 보유된 모든 커서가 닫힙니다.

작업 단위(UOW)가 ROLLBACK문으로 종료되는 경우, 모든 개방 커서는 닫히고, 작업 단위 중에 획득된 모든 잠금이 해제되며, 해당 작업 단위에서 완료된 작업에 종속적인 준비된 모든 명령문들이 제거됩니다.

예를 들어, TEMPL 테이블에는 1000개 항목이 들어 있습니다. 모든 사원의 급여 컬럼을 갱신하고, 100개 행을 갱신할 때마다 COMMIT문을 발행한다고 가정하십시오.

1. WITH HOLD 옵션을 사용하여 커서를 선언합니다.

```
EXEC SQL DECLARE EMPLUPDT CURSOR WITH HOLD FOR
  SELECT EMPNO, LASTNAME, PHONENO, JOBCODE, SALARY
  FROM TEMPL FOR UPDATE OF SALARY
```

2. 커서를 열고 결과 테이블에서 한 번에 한 행의 데이터를 가져옵니다.

```
EXEC SQL OPEN EMPLUPDT
  .
  .
  .

EXEC SQL FETCH EMPLUPDT
  INTO :upd_emp, :upd_lname, :upd_tele, :upd_jobcd, :upd_wage,
```

3. 행을 갱신하거나 삭제하려면, WHERE CURRENT OF 옵션과 함께 UPDATE문 또는 DELETE문을 사용하십시오. 예를 들어, 현재 행을 갱신하기 위해 프로그램이 다음을 실행할 수 있습니다.

```
EXEC SQL UPDATE TEMPL SET SALARY = :newsalary
  WHERE CURRENT OF EMPLUPDT
```

4. COMMIT가 발행된 후 다른 행을 갱신하려면 우선 FETCH를 발행해야 합니다.

응용프로그램이 다음과 같은 경우 FETCH문 또는 CLOSE문에서 리턴될 수 있는 SQLCODE -501(SQLSTATE 24501)을 검출 및 처리하려면 사용자 응용프로그램에 코드를 포함시켜야 합니다.

- WITH HOLD 선언된 커서 사용
- 하나 이상의 작업 단위를 실행하고 해당 작업 단위 경계(COMMIT WORK) 내에서 WITH HOLD 커서를 열린 상태로 둡니다.

종속된 테이블을 제거함으로써 응용프로그램이 패키지를 무효화할 경우 패키지는 동적으로 리바인드됩니다. 이런 경우, 데이터베이스 관리 프로그램이 커서를 닫으므로, FETCH문 또는 CLOSE문에 대해 SQLCODE -501(SQLSTATE 24501)이 리턴됩니다. 이런 상황에서 SQLCODE -501(SQLSTATE 24501)을 처리하는 방법은 커서로부터 행을 페치할 것인지에 따라 달라집니다.

- 커서로부터 행을 페치하려면 커서를 열고 FETCH문을 수행하십시오. 그러나, OPEN문은 커서 위치를 다시 시작 부분으로 가져갑니다. 이전에 COMMIT WORK문에 있던 위치는 소실됩니다.
- 커서에서 행을 페치하지 않으려면 커서에 대해 더이상의 SQL 요청을 발행하지 마십시오.

WITH RELEASE 옵션: 응용프로그램이 WITH RELEASE 옵션을 사용하여 커서를 닫을 때 DB2는 커서가 보유하고 있는 모든 READ 잠금을 해제하려 합니다. 커서는 계속해서 WRITE 잠금만 보유합니다. 응용프로그램이 RELEASE 옵션을 사용하지 않고 커서를 닫으면, 작업 단위(UOW)가 완료될 때 READ 및 WRITE 잠금이 해제됩니다.

예: 커서 프로그램

이 샘플 프로그램은 커서를 정의 및 사용하는 SQL문을 보여 줍니다. 커서는 정적 SQL을 사용하여 처리됩니다. 샘플은 다음과 같은 프로그래밍 언어로 사용할 수 있습니다:

C	cursor.sqc
Java	Cursor.sqlj
COBOL	cursor.sqb

REXX가 정적 SQL을 지원하지 않으므로 샘플은 제공되지 않습니다. 커서를 동적으로 처리하는 REXX 예에 대해서는 152 페이지의『예: 동적 SQL 프로그램』의 내용을 참조하십시오.

커서 프로그램 작업 방법

1. **커서 선언.** DECLARE CURSOR문은 커서 c1을 조회에 연관시킵니다. 조회는 FETCH문을 사용하여 응용프로그램이 검색하는 행을 식별합니다. staff의 job 필드는 결과 테이블에 지정되지 않더라도 갱신이 가능하도록 정의됩니다.
2. **커서 열기.** 커서 c1이 열립니다. 데이터베이스 관리 프로그램이 결과 테이블을 조회하고 빌드하도록 합니다. 커서는 첫번째 행 *앞에* 위치하게 됩니다.
3. **행 검색.** FETCH문은 커서를 다음 행에 위치시키고 행의 내용을 호스트 변수로 이동시킵니다. 이 행은 *현재* 행이 됩니다.
4. **커서 닫기.** CLOSE문이 발행됩니다. 커서와 연관된 자원들을 릴리스합니다. 하지만 커서가 또 다시 열릴 수 있습니다.

CHECKERR 매크로/함수는 오류 체크 유틸리티입니다. 이 오류 체크 유틸리티의 위치는 사용된 프로그래밍 언어에 따라 달라질 수 있습니다.

C
DB2 API를 호출하는 C 프로그램에 대해서는 utilapi.c에서 sqlInfoPrint 함수는 utilapi.h에서 API_SQL_CHECK로서 참조됩니다. C embedded SQL 프로그램에 대해서는 utilemb.sqc에서 sqlInfoPrint 함수는 utilemb.h에서 EMB_SQL_CHECK로서 참조됩니다.

Java
SQL 오류는 SQLException으로서 드로우되고, 응용프로그램의 catch 블록에서 처리됩니다.

COBOL
CHECKERR은 checkerr.cbl라고 하는 외부 프로그램입니다.

FORTRAN
CHECKERR은 util.f 파일에 있는 서브루틴입니다.

이 오류 체크 유틸리티에 대한 소스 코드에 대해서는 137 페이지의 『예제 프로그램에서 GET ERROR MESSAGE 사용』의 내용을 참조하십시오.

C 예: CURSOR.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[])
{

   EXEC SQL BEGIN DECLARE SECTION;
      char   pname[10];
      short  dept;
      char userid[9];
      char passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: CURSOR \n" );

   if (argc == 1)
   {
      EXEC SQL CONNECT TO sample;
  EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3)
   {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else
   {
      printf ("\nUSAGE: cursor [userid passwd]\n\n");
      return 1;
   } /* endif */

   EXEC SQL DECLARE c1 CURSOR FOR   1
            SELECT name, dept FROM staff WHERE job='Mgr'
            FOR UPDATE OF job;

   EXEC SQL OPEN c1;   2
   EMB_SQL_CHECK("OPEN CURSOR");

   do
   {
```

```
      EXEC SQL FETCH c1 INTO :pname, :dept;   3
      if (SQLCODE != 0) break;

      printf( "%-10.10s in dept. %2d will be demoted to Clerk\n",
         pname, dept );
   } while ( 1 );

   EXEC SQL CLOSE c1;  4
   EMB_SQL_CHECK("CLOSE CURSOR");

   EXEC SQL ROLLBACK;
   EMB_SQL_CHECK("ROLLBACK");
   printf( "\nOn second thought -- changes rolled back.\n" );

   EXEC SQL CONNECT RESET;
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}
/* end of program : CURSOR.SQC */
```

Java 예: Cursor.sqlj

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator CursorByName(String name, short dept) ;
#sql iterator CursorByPos(String, short ) ;

class Cursor
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
    }
    catch (Exception e)
    {   System.out.println ("\n  Error loading DB2 Driver...\n");
      System.out.println (e);
      System.exit(1);
    }
  }

  public static void main(String argv[])
  {   try
    {   System.out.println ("  Java Cursor Sample");

      String url = "jdbc:db2:sample";       // URL is jdbc:db2:dbname
      Connection con = null;

      // Set the connection
      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception("\nUsage: java Cursor [username password]\n");
      }

      // Set the default context
      DefaultContext ctx = new DefaultContext(con);
      DefaultContext.setDefaultContext(ctx);

      // Enable transactions
      con.setAutoCommit(false);

      // Using cursors
      try
      {   CursorByName cursorByName;
        CursorByPos  cursorByPos;

        String name = null;
        short  dept=0;
```

```
      // Using the JDBC ResultSet cursor method
      System.out.println("\nUsing the JDBC ResultSet cursor method");
      System.out.println(" with a 'bind by name' cursor ...\n");

      #sql cursorByName = {
            SELECT name, dept FROM staff WHERE job='Mgr' };  1
      while (cursorByName.next())  2
      {   name = cursorByName.name();  3
        dept = cursorByName.dept();

        System.out.print (" name= " + name);
        System.out.print (" dept= " + dept);
        System.out.print ("\n");
      }
      cursorByName.close();  4


      // Using the SQLJ iterator cursor method
      System.out.println("\nUsing the SQLJ iterator cursor method");
      System.out.println(" with a 'bind by position' cursor ...\n");

      #sql cursorByPos = {
             SELECT name, dept FROM staff WHERE job='Mgr' };  1  2
      while (true)
      {   #sql { FETCH :cursorByPos INTO :name, :dept };  3
        if (cursorByPos.endFetch()) break;

        System.out.print (" name= " + name);
        System.out.print (" dept= " + dept);
        System.out.print ("\n");
      }
      cursorByPos.close();  4
    }
    catch( Exception e )
    {   throw e;
    }
    finally
    {   // Rollback the transaction
      System.out.println("\nRollback the transaction...");
      #sql { ROLLBACK };
      System.out.println("Rollback done.");
    }
  }
  catch( Exception e )
  {   System.out.println (e);
  }
 }
}
```

COBOL 예: CURSOR.SQB

```
Identification Division.
Program-ID. "cursor".

Data Division.
Working-Storage Section.

    copy "sqlenv.cbl".
    copy "sql.cbl".
    copy "sqlca.cbl".

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 pname             pic x(10).
77 dept              pic s9(4) comp-5.
01 userid            pic x(8).
01 passwd.
  49 passwd-length   pic s9(4) comp-5 value 0.
  49 passwd-name     pic x(18).
    EXEC SQL END DECLARE SECTION END-EXEC.

77 errloc          pic x(80).

Procedure Division.
Main Section.
    display "Sample COBOL program: CURSOR".

    display "Enter your user id (default none): "
         with no advancing.
    accept userid.

    if userid = spaces
      EXEC SQL CONNECT TO sample END-EXEC
    else
      display "Enter your password : " with no advancing
      accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.
    inspect passwd-name tallying passwd-length for characters
       before initial " ".

    EXEC SQL CONNECT TO sample USER :userid USING :passwd
       END-EXEC.
    move "CONNECT TO" to errloc.
    call "checkerr" using SQLCA errloc.

    EXEC SQL DECLARE c1 CURSOR FOR                   1
             SELECT name, dept FROM staff
             WHERE job='Mgr'
             FOR UPDATE OF job END-EXEC.

    EXEC SQL OPEN c1 END-EXEC.                 2
    move "OPEN CURSOR" to errloc.
    call "checkerr" using SQLCA errloc.

    perform Fetch-Loop thru End-Fetch-Loop
       until SQLCODE not equal 0.

    EXEC SQL CLOSE c1 END-EXEC.                4
    move "CLOSE CURSOR" to errloc.
    call "checkerr" using SQLCA errloc.

    EXEC SQL ROLLBACK END-EXEC.
    move "ROLLBACK" to errloc.
    call "checkerr" using SQLCA errloc.
```

```
    DISPLAY "On second thought -- changes rolled back.".

    EXEC SQL CONNECT RESET END-EXEC.
    move "CONNECT RESET" to errloc.
    call "checkerr" using SQLCA errloc.
End-Main.
    go to End-Prog.

Fetch-Loop Section.
    EXEC SQL FETCH c1 INTO :PNAME, :DEPT END-EXEC.      3
    if SQLCODE not equal 0
       go to End-Fetch-Loop.
    display pname, " in dept. ", dept,
       " will be demoted to Clerk".
End-Fetch-Loop. exit.

End-Prog.
    stop run.
```

검색된 데이터 갱신 및 삭제

커서가 참조하는 행을 갱신 및 삭제하는 것이 가능합니다. 행을 갱신 가능하도록 하기 위해 커서에 해당되는 조회는 읽기 전용이어서는 안됩니다. 조회를 갱신 가능하거나 삭제가능하도록 하는 것에 대한 설명은 *SQL 참조서*의 내용을 참조하십시오.

검색된 데이터 갱신

커서로 갱신하려면 UPDATE문에서 WHERE CURRENT OF절을 사용하십시오. 결과 테이블의 일부 컬럼을 갱신할 시스템을 알아보려면 FOR UPDATE절을 사용하십시오. fullselect에 있지 않은 채 FOR UPDATE에 컬럼을 지정할 수 있습니다. 따라서, 커서에 의해 명시적으로 검색되지 않는 컬럼을 갱신할 수 있습니다. FOR UPDATE절이 컬럼 이름 없이 지정될 경우, 외부 fullselect의 첫번째 FROM 절에서 식별되는 뷰 또는 테이블의 모든 컬럼들이 갱신 가능하다고 간주됩니다. FOR UPDATE절에 필요한 것보다 많은 컬럼을 명명하지 마십시오. FOR UPDATE절에서 추가 컬럼을 명명하면 DB2가 데이터에 액세스할 때 효율성이 떨어지는 경우가 있습니다.

검색된 데이터 삭제

커서로 삭제하는 것은 DELETE문에서 WHERE CURRENT OF절을 사용하여 수행됩니다. 일반적으로 FOR UPDATE절은 커서의 현재 행 삭제에 필요하지 않습니다. 유일한 예외는, LANGLEVEL이 SAA1으로 설정되었고 BLOCKING ALL로 바인드된 응용프로그램에서 SELECT문이나 DELETE문에 대해 동적 SQL을 사용할 때(동적 SQL에 대해서는 145 페이지의 『제5장 동적 SQL 프로그램 작성』 참조) 발생합니다. 이런 경우 FOR UPDATE절은 SELECT문에서 필수적입니다. 사전 처리 컴파일러 옵션에 대해서는 *Command Reference*의 내용을 참조하십시오.

DELETE문은 삭제할 커서에 의해 참조되는 행을 삭제합니다. 그 결과 커서는 *다음* 행 앞에 그대로 위치하며, FETCH문은 추가 WHERE CURRENT OF 조작이 커서에 대해 수행되기 전에 실행되어야 합니다.

커서 유형

커서는 다음 세 범주로 나뉩니다.

읽기 전용

커서의 행들은 읽을 수만 있고 갱신은 불가능합니다. 읽기 전용 커서는 응용프로그램이 데이터를 읽기만 하고 수정은 하지 않을 때 사용됩니다. 읽기 전용 select문에 기초한 경우에만 커서가 고려됩니다. 갱신 불가 결과 테이블을 정의하는 select문에 대해서는 107 페이지의『검색된 데이터 갱신』에 있는 규칙을 참조하십시오.

읽기 전용 커서에는 성능상의 이점이 있을 수 있습니다. 읽기 전용 커서에 대한 세부사항은 *관리 안내서: 구현*의 내용을 참조하십시오.

갱신가능

커서의 행들은 갱신할 수 있습니다. 갱신가능 커서는 커서의 행들이 페치될 때 응용프로그램이 데이터를 수정할 때 사용됩니다. 지정된 조회는 한 테이블 또는 뷰만 참조할 수 있습니다. (LANGLEVEL MIA 사전 처리 컴파일 옵션을 사용하지 않는 한) 갱신될 각 컬럼을 명명하여, 조회에는 FOR UPDATE절도 포함되어야 합니다.

명확한

커서는 정의 또는 문맥으로부터만 읽거나 갱신 가능하도록 결정할 수 없습니다. 그렇지 않으면 읽기 전용으로 간주될 커서를 변경하는 데 사용할 동적 SQL문이 발견될 때 이와 같은 현상이 발생합니다.

불명확한 커서는 사전 처리 컴파일 또는 바인딩될 때 BLOCKING ALL 옵션이 지정되는 경우에만 읽기 전용으로 취급됩니다. 그렇지 않으면 갱신 가능으로 간주됩니다.

주: 동적으로 처리되는 커서는 항상 명확합니다.

커서가 읽기 전용, 갱신가능 또는 명확 중에서 어느 것인지 결정하는 데 사용되는 완벽한 기준 목록은 *SQL 참조서*의 내용을 참조하십시오.

예: OPENFTCH 프로그램

이 예는 커서를 사용하여 테이블로부터 커서를 선택하여 열고, 테이블로부터 행을 페치합니다. 페치되는 각 행에 대해서 (간단한 기준에 의해) 행을 삭제할 것인지 또는 갱신할 것인지 결정합니다. 샘플은 다음과 같은 프로그래밍 언어로 사용할 수 있습니다:

C
openftch.sqc

Java
Openftch.sqlj 및 OpF_Curs.sqlj

COBOL
openftch.sqb

REXX 언어는 정적 SQL을 지원하지 않으므로 샘플은 제공되지 않습니다.

OPENFTCH 프로그램 작업 방법

1. **커서 선언.** DECLARE CURSOR문은 커서 `c1`을 조회에 연관시킵니다. 조회는 FETCH문을 사용하여 응용프로그램이 검색하는 행을 식별합니다. `staff`의 `job` 필드는 결과 테이블에 지정되지 않더라도 갱신이 가능하도록 정의됩니다.
2. **커서 열기.** 커서 `c1`이 열립니다. 데이터베이스 관리 프로그램이 결과 테이블을 조회하고 빌드하도록 합니다. 커서는 첫번째 행 *앞에* 위치하게 됩니다.
3. **행 검색.** FETCH문은 커서를 다음 행에 위치시키고 행의 내용을 호스트 변수로 이동시킵니다. 이 행은 *현재* 행이 됩니다.
4. **현재 행 갱신 또는 삭제.** 현재 행은 `FETCH`문으로 리턴되는 `dept` 값에 따라 갱신되거나 삭제됩니다.

   UPDATE가 수행되면 UPDATE문이 현재 행의 위치를 변경시키지 않으므로, 커서는 이 행에 남아 있습니다.

   DELETE문이 수행되면, *현재* 행이 삭제되므로 다른 상황이 발생합니다. 이는 *다음* 행 앞에 위치하는 것과 같으며, FETCH문은 추가 WHERE CURRENT OF 조작이 수행되기 전에 발행되어야 합니다.
5. **커서 닫기.** CLOSE문이 발행됩니다. 커서와 연관된 자원들을 릴리스합니다. 하지만 커서가 또 다시 열릴 수 있습니다.

`CHECKERR` 매크로/함수는 오류 체크 유틸리티입니다. 이 오류 체크 유틸리티의 위치는 사용된 프로그래밍 언어에 따라 달라질 수 있습니다.

C DB2 API를 호출하는 C 프로그램에 대해서는 utilapi.c에서 sqlInfoPrint 함수는 utilapi.h에서 API_SQL_CHECK로서 참조됩니다. C embedded SQL 프로그램에 대해서는 utilemb.sqc에서 sqlInfoPrint 함수는 utilemb.h에서 EMB_SQL_CHECK로서 참조됩니다.

Java SQL 오류는 SQLException으로서 드로우되고, 응용프로그램의 catch 블록에서 처리됩니다.

COBOL CHECKERR은 checkerr.cbl라고 하는 외부 프로그램입니다.

이 오류 체크 유틸리티에 대한 소스 코드에 대해서는 137 페이지의『예제 프로그램에서 GET ERROR MESSAGE 사용』의 내용을 참조하십시오.

C 예: OPENFTCH.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[])
{

   EXEC SQL BEGIN DECLARE SECTION;
      char   pname[10];
      short  dept;
      char userid[9];
      char passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: OPENFTCH\n" );

   if (argc == 1)
   {
      EXEC SQL CONNECT TO sample;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3)
   {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else
   {
      printf ("\nUSAGE: openftch [userid passwd]\n\n");
      return 1;
   } /* endif */

   EXEC SQL DECLARE c1 CURSOR FOR   1
            SELECT name, dept FROM staff WHERE job='Mgr'
            FOR UPDATE OF job;

   EXEC SQL OPEN c1;   2
   EMB_SQL_CHECK("OPEN CURSOR");

   do
   {
```

```
      EXEC SQL FETCH c1 INTO :pname, :dept;   3
      if (SQLCODE != 0) break;

      if (dept > 40)
      {
         printf( "%-10.10s in dept. %2d will be demoted to Clerk\n",
            pname, dept );
         EXEC SQL UPDATE staff SET job = 'Clerk'   4
            WHERE CURRENT OF c1;
         EMB_SQL_CHECK("UPDATE STAFF");
      }
      else
      {
         printf ("%-10.10s in dept. %2d will be DELETED!\n",
            pname, dept);
         EXEC SQL DELETE FROM staff WHERE CURRENT OF c1;
         EMB_SQL_CHECK("DELETE");
      } /* endif */
   } while ( 1 );

   EXEC SQL CLOSE c1;   5
   EMB_SQL_CHECK("CLOSE CURSOR");

   EXEC SQL ROLLBACK;
   EMB_SQL_CHECK("ROLLBACK");
   printf( "\nOn second thought -- changes rolled back.\n" );

   EXEC SQL CONNECT RESET;
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}
/* end of program : OPENFTCH.SQC */
```

Java 예: Openftch.sqlj

OpF_Curs.sqlj

```
//  PURPOSE : This file, named OpF_Curs.sqlj, contains the definition
//            of the class OpF_Curs used in the sample program Openftch.

import sqlj.runtime.ForUpdate;
#sql public iterator OpF_Curs implements ForUpdate (String, short);
```

Openftch.sqlj

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

class Openftch
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
    }
    catch (Exception e)
    {   System.out.println ("\n  Error loading DB2 Driver...\n");
      System.out.println (e);
      System.exit(1);
    }
  }

  public static void main(String argv[])
  {   try
    {   System.out.println ("  Java Openftch Sample");

      String url = "jdbc:db2:sample";       // URL is jdbc:db2:dbname
      Connection con = null;

      // Set the connection
      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception(
```

```
            "\nUsage: java Openftch [username password]\n");
} // if - else if - else

// Set the default context
DefaultContext ctx = new DefaultContext(con);
DefaultContext.setDefaultContext(ctx);

// Enable transactions
con.setAutoCommit(false);

// Executing SQLJ positioned update/delete statements.
try
{   OpF_Curs forUpdateCursor;

  String name = null;
  short  dept=0;


  #sql forUpdateCursor =
  {   SELECT name, dept
          FROM staff
          WHERE job='Mgr'
  }; // #sql              1 2

  while (true)
  {   #sql
    {   FETCH :forUpdateCursor
            INTO :name, :dept
    }; // #sql                            3
    if (forUpdateCursor.endFetch()) break;

    if (dept > 40)
    {   System.out.println (
            name + " in dept. "
            + dept + " will be demoted to Clerk");
      #sql
      {   UPDATE staff SET job = 'Clerk'
          WHERE CURRENT OF :forUpdateCursor
      }; // #sql                          4
    }
    else
    {   System.out.println (
            name + " in dept. " + dept
            + " will be DELETED!");
      #sql
      {   DELETE FROM staff
          WHERE CURRENT OF :forUpdateCursor
      }; // #sql
```

```
        } // if - else
      }
      forUpdateCursor.close();   5
    }
    catch( Exception e )
    {   throw e;
    }
    finally
    {   // Rollback the transaction
      System.out.println("\nRollback the transaction...");
      #sql { ROLLBACK };
      System.out.println("Rollback done.");
    } // try - catch - finally
  }
  catch( Exception e )
  {   System.out.println (e);
  } // try - catch
} // main
} // class Openftch
```

COBOL 예: OPENFTCH.SQB

```
       Identification Division.
       Program-ID. "openftch".

       Data Division.
       Working-Storage Section.

           copy "sqlca.cbl".

           EXEC SQL BEGIN DECLARE SECTION END-EXEC.
       01 pname             pic x(10).
       01 dept              pic s9(4) comp-5.
       01 userid            pic x(8).
       01 passwd.
         49 passwd-length   pic s9(4) comp-5 value 0.
         49 passwd-name     pic x(18).
           EXEC SQL END DECLARE SECTION END-EXEC.

       77 errloc          pic x(80).

       Procedure Division.
       Main Section.
           display "Sample COBOL program: OPENFTCH".

      * Get database connection information.
           display "Enter your user id (default none): "
                with no advancing.
           accept userid.

           if userid = spaces
             EXEC SQL CONNECT TO sample END-EXEC
           else
             display "Enter your password : " with no advancing
             accept passwd-name.

      * Passwords in a CONNECT statement must be entered in a VARCHAR format
      * with the length of the input string.
           inspect passwd-name tallying passwd-length for characters
              before initial " ".

           EXEC SQL CONNECT TO sample USER :userid USING :passwd
               END-EXEC.
           move "CONNECT TO" to errloc.
           call "checkerr" using SQLCA errloc.

           EXEC SQL DECLARE c1 CURSOR FOR          1
                    SELECT name, dept FROM staff
                    WHERE job='Mgr'
                    FOR UPDATE OF job END-EXEC.

           EXEC SQL OPEN c1 END-EXEC          2
           move "OPEN" to errloc.
           call "checkerr" using SQLCA errloc.

      * call the FETCH and UPDATE/DELETE loop.
           perform Fetch-Loop thru End-Fetch-Loop
              until SQLCODE not equal 0.

           EXEC SQL CLOSE c1 END-EXEC.          5
           move "CLOSE" to errloc.
           call "checkerr" using SQLCA errloc.

           EXEC SQL ROLLBACK END-EXEC.
           move "ROLLBACK" to errloc.
           call "checkerr" using SQLCA errloc.
```

```
    display "On second thought -- changes rolled back.".

    EXEC SQL CONNECT RESET END-EXEC.
    move "CONNECT RESET" to errloc.
    call "checkerr" using SQLCA errloc.
End-Main.
    go to End-Prog.

Fetch-Loop Section.
    EXEC SQL FETCH c1 INTO :pname, :dept END-EXEC.         3
    if SQLCODE not equal 0
       go to End-Fetch-Loop.

    if dept greater than 40
       go to Update-Staff.

Delete-Staff.
    display pname, " in dept. ", dept,
       " will be DELETED!".

    EXEC SQL DELETE FROM staff WHERE CURRENT OF c1 END-EXEC.
    move "DELETE" to errloc.
    call "checkerr" using SQLCA errloc.

    go to End-Fetch-Loop.

Update-Staff.
    display pname, " in dept. ", dept,
       " will be demoted to Clerk".

    EXEC SQL UPDATE staff SET job = 'Clerk'            4
             WHERE CURRENT OF c1 END-EXEC.
    move "UPDATE" to errloc.
    call "checkerr" using SQLCA errloc.

End-Fetch-Loop. exit.

End-Prog.
    stop run.
```

고급 화면이동 기술

고급 화면이동 기술에 대해 다음과 같은 주제가 이 절에서 논의됩니다.

- 이미 검색된 데이터의 화면이동
- 데이터 사본 보관
- 데이터 두 번째 검색
- 테이블 끝에 위치 설정
- 이전에 검색된 데이터 갱신

이미 검색된 데이터의 화면이동

응용프로그램이 데이터베이스로부터 데이터를 검색할 때 FETCH문은 데이터 앞으로 화면 이동하도록 하지만, 데이터베이스 관리 프로그램은 데이터 뒤로 화면 이동하도록 하는 Embedded SQL문이 없습니다(backward FETCH와 동일). 그러나, DB2 CLI 및 Java는 읽기 전용 화면 이동 커서를 통해 backward FETCH를 지원합니다. 화면이동 커서에 대한 세부사항은 *CLI Guide and Reference* 및 715 페이지의 『Java 응용프로그램 및 애플릿 작성』의 내용을 참조하십시오. Embedded SQL 응용프로그램의 경우, 다음 기술을 사용하여 검색된 데이터를 화면이동할 수 있습니다.

1. 페치된 데이터 사본을 보유한 채 일부 프로그래밍 기술을 이용하여 화면 이동을 합니다.
2. 일반적으로 두 번째 SELECT문으로 SQL을 사용하여 다시 데이터를 검색합니다.

이 옵션들에 대해서는 다음에서 보다 상세히 다룰 것입니다:

- 데이터 사본 보관
- 데이터 두 번째 검색

데이터 사본 보관

응용프로그램은 페치한 데이터를 가상 저장영역에 저장할 수 있습니다. 데이터가 가상 저장영역에 맞지 않으면 응용프로그램은 데이터를 임시 파일에 기록할 수 없습니다. 트랜잭션에 의해 중간에 데이터베이스에 있는 데이터가 변경되었어도, 이 접근 방식의 결과로 사용자는 뒤로 화면이동하면서 항상 페치된 것과 같은 데이터를 볼 수 있습니다.

반복 읽기(RR)의 분리 레벨을 사용하여 트랜잭션으로부터 검색되는 데이터는 커서를 닫고 열어 다시 검색할 수 있습니다. 기타 응용프로그램에서는 사용자의 결과 세트에 있는 데이터를 갱신하지 못합니다. 분리 레벨 및 잠금은 사용자의 데이터 갱신 방법에 영향을 줄 수 있습니다.

데이터 두 번째 검색

이 기술은 다시 데이터를 보고자 하는 순서에 따라 달라집니다.

- 시작부터 검색
- 중간부터 검색
- 두 번째 결과 테이블의 행 순서
- 역순 검색

시작부터 검색

시작부터 다시 데이터를 검색하려면 활동중인 커서를 닫고 다시 열기만 하면 됩니다. 이러한 조치는 커서를 결과 테이블의 시작 부분에 둡니다. 그러나, 응용프로그램이 테이블에 대한 잠금을 보유하지 않는 한, 다른 테이블들은 변경되었을 수 있으므로 결과 테이블의 첫번째 행이었던 것은 더이상 첫번째 행이 아닙니다.

중간부터 검색

결과 테이블 중간 부분에서 데이터를 두 번째 검색하려면 두 번째 SELECT문을 실행하고 명령문에 대해 두 번째 커서를 선언하십시오. 예를 들어, 첫번째 SELECT 문이 다음과 같았다고 가정하십시오.

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  ORDER BY DEPTNO
```

이제DEPTNO = 'M95'로 시작하는 행으로 리턴하여 이 지점에서 순차적으로 페치한다고 가정합니다. 다음을 코드화하십시오.

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  AND DEPTNO >= 'M95'
  ORDER BY DEPTNO
```

이 명령문은 원하는 장소에 커서를 둡니다.

두 번째 결과 테이블의 행 순서

두 번째 결과 테이블의 행들이 첫번째 결과 테이블에서와 같은 수서로 표시되지 않을 수도 있습니다. 데이터베이스 관리 프로그램은 SELECT문이 ORDER BY를 사용하지 않는 한, 행의 순서를 중요하게 여기지 않습니다. 따라서, 동일한 DEPTNO 값을 가진 행이 여러 개 있을 경우 두 번째 SELECT문은 첫번째와는 다른 순서로 검색할 수 있습니다. 확실한 한가지는 ORDER BY DEPTNO절에 의해 요구될 경우 모두 부서 번호순으로 검색되는 것입니다.

동일한 호스트 변수를 가지고 동일한 SQL문을 두 번 실행할 경우에도 순서가 달라질 수 있습니다. 예를 들어, 실행 사이에 카탈로그의 통계가 갱신되거나, 색인이 작성 또는 제거될 수 있습니다. 그 다음에는 다시 SELECT문을 실행할 수 있습니다.

첫번째 SELECT에는 없었던 술어가 두 번째 SELECT에 있을 경우 순서가 변경될 가능성이 더 높아집니다. 데이터베이스 관리 프로그램은 새 술어에서 색인을 사용하도록 선택할 수 있습니다. 예를 들어, 위 예에서 첫번째 명령문의 경우 LOCATION에 대해, 두 번째 명령문의 경우 DEPTNO에 대해 색인을 선택할 수 있습니다. 행은 색인 키 순서로 페치되므로 두 번째 순서는 첫번째 순서와 같지 않아도 됩니다.

또다시, 두 개의 유사한 SELECT문을 실행할 때 통계가 변경되지 않고 색인이 작성되거나 제거되지 않아도 행 순서가 달라질 수 있습니다. 이 예에서 LOCATION에 대해 다른 값들이 많이 있을 경우 데이터베이스 관리 프로그램은 두 명령문 모두에 대해 LOCATION에서 색인을 선택할 수 있습니다. 그러나 두 번째 명령문의 DEPTNO 값을 다음과 같이 변경할 경우 데이터베이스 관리 프로그램은 DEPTNO에 대한 색인을 선택할 수 있습니다.

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  AND DEPTNO >= 'Z98'
  ORDER BY DEPTNO
```

SQL문의 형식과 이 명령문의 값들 사이에는 미묘한 관계가 있으므로, 두 개의 순서가 ORDER BY절에 의해 고유하게 결정되지 않는 한 두 개의 서로 다른 SQL문이 같은 순서로 행을 리턴한다고 가정해서는 안 됩니다.

역순 검색

행들이 오름차순으로 정렬되는 것이 기본값입니다. DEPTNO의 각 값에 대해 한 행만 있는 경우 다음 명령문은 행에 고유한 오름차순을 지정합니다.

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  ORDER BY DEPTNO
```

같은 행들을 역순으로 검색하려면 다음 명령문에서처럼 내림차순으로 지정하십시오.

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  ORDER BY DEPTNO DESC
```

두 번째 명령문의 커서는 첫번째 명령문의 커서와 정반대로 행을 검색합니다. 검색 순서는 첫번째 명령문이 고유한 절차를 지정할 경우에만 효력을 발휘합니다.

행을 역순으로 검색하려면 DEPTNO 컬럼에 두 개의 색인, 즉 하나는 오름차순으로 다른 하나는 내림차순으로 하는 것이 유용할 것입니다.

테이블 끝에 위치 설정

데이터베이스 관리 프로그램은 테이블에 저장된 데이터에 대한 순서를 보장하지 못하므로 테이블의 끝은 정의되지 않습니다. 그러나, 순서는 SQL문의 결과에 정의됩니다.

```
SELECT * FROM DEPARTMENT
  ORDER BY DEPTNO DESC
```

이 예에서 다음 명령문은 최고의 DEPTNO 값을 가진 행에 커서를 둡니다.

```
SELECT * FROM DEPARTMENT
  WHERE DEPTNO =
  (SELECT MAX(DEPTNO) FROM DEPARTMENT)
```

그러나, 여러 행의 값이 같을 경우 커서는 첫번째 행에 위치합니다.

이전에 검색된 데이터 갱신

뒤로 화면이동하거나 이전에 검색된 데이터를 갱신하려면 118 페이지의『이미 검색된 데이터의 화면이동』 및 107 페이지의『검색된 데이터 갱신』에 설명되어 있는 기술들을 조합하여 사용해 보십시오. 다음 중 하나를 수행할 수 있습니다.

1. 두 번째 커서가 갱신될 데이터에 있고 SELECT문이 제한된 요소를 사용하지 않는 경우 커서에 의해 제어되는 UPDATE문을 사용할 수 있습니다. WHERE CURRENT OF절에서 두 번째 커서를 명명하십시오.
2. 다른 경우, 행의 모든 값들을 명명하는 WHERE절에서 UPDATE를 사용하거나, 테이블의 기본 키를 지정하십시오. 서로 다른 변수 값들로 한 명령문을 여러 번 실행할 수 있습니다.

예: UPDAT 프로그램

UPDAT 프로그램은 동적 SQL을 사용하여 SAMPLE 데이터베이스의 STAFF 테이블에 액세스한 후, 모든 관리자를 사원으로 변경합니다. 그러면 프로그램은 작업 단위(UOW)를 구간 복원하여 변경사항을 복원시킵니다. 샘플은 다음과 같은 프로그래밍 언어로 사용할 수 있습니다:

C	updat.sqc
Java	Updat.sqlj
COBOL	updat.sqb
REXX	updat.cmd

UPDAT 프로그램 작업 방법

1. **SQLCA 구조 정의** INCLUDE SQLCA문은 SQLCA 구조를 정의 및 선언하고, 구조 내의 요소로서 SQLCODE를 정의합니다. SQLCA의 SQLCODE 필드는 SQL문과 데이터베이스 관리 프로그램 API 호출이 실행된 후 데이터베이스 관리 프로그램에 의해 오류 정보로 갱신됩니다.

   Java 응용프로그램은 SQLException 오브젝트에 대해 정의된 메소드를 통해 SQLCODE 및 SQLSTATE에 액세스하므로 동등한 "include SQLCA"문이 필요하지 않습니다.

   REXX 응용프로그램에는 응용프로그램 사용을 위해 사전 정의된, SQLCA라는 SQLCA 구조의 발생이 있습니다. 이 발생은 응용프로그램의 정의 없이 참조될 수 있습니다.
2. **호스트 변수 정의.** BEGIN DECLARE SECTION 및 END DECLARE SECTION문은 호스트 변수 선언을 분리합니다. 호스트 변수를 사용하여 데이

터를 데이터베이스 관리 프로그램으로 전달하거나 데이터베이스 관리 프로그램으로부터 데이터를 전달합니다. 호스트 변수는 SQL문에서 참조될 때 콜론(:)으로 시작됩니다.

Java 및 REXX 응용프로그램은 LOB 파일 참조 변수 및 위치 지정자의 경우(REXX)를 제외하고, 호스트 변수를 선언할 필요가 없습니다. 호스트 변수 데이터 유형 및 크기는 변수가 참조될 때 런타임시 결정됩니다.

3. **데이터베이스에 연결.** 프로그램은 `sample` 데이터베이스에 연결하여 이 데이터베이스에 대한 공유 액세스를 요청합니다. (START DATABASE MANAGER API 호출이나 `db2start` 명령이 발행되었다고 전제합니다.) 공유 액세스를 사용하여 동일한 데이터베이스에 연결하는 다른 프로그램들에게도 액세스가 권한 부여됩니다.
4. **UPDATE SQL문 실행.** SQL문은 호스트 변수를 사용하여 *정적*으로 실행됩니다. `staff` 테이블의 `job` 컬럼은 호스트 변수 값으로 설정됩니다. 여기서 `job` 컬럼에는 `Mgr` 값이 있습니다.
5. **DELETE SQL문 실행** SQL문은 호스트 변수 사용과 함께 *정적*으로 실행됩니다. 지정된 호스트 변수와 같은 `job` 컬럼 값(`jobUpdate/job-update/job_update`)이 있는 모든 행들은 삭제됩니다.
6. **INSERT SQL문 실행** 행이 `STAFF` 테이블에 삽입됩니다. 이 삽입으로 이 SQL문의 실행 전에 설정된 호스트 변수 사용이 구현됩니다.
7. **트랜잭션 종료.** ROLLBACK문으로 *작업 단위(UOW)*를 종료합니다. 이전에 실행된 SQL문 결과는 COMMIT문을 사용하여 영구적으로 되거나, ROLLBACK문을 사용하여 실행 취소됩니다. *작업 단위(UOW)* 내의 모든 SQL문이 영향을 받습니다.

`CHECKERR` 매크로/함수는 오류 체크 유틸리티입니다. 이 오류 체크 유틸리티의 위치는 사용된 프로그래밍 언어에 따라 달라질 수 있습니다.

C DB2 API를 호출하는 C 프로그램에 대해서는 utilapi.c에서 `sqlInfoPrint` 함수는 utilapi.h에서 `API_SQL_CHECK`로서 참조됩니다. C embedded SQL 프로그램에 대해서는 utilemb.sqc에서 `sqlInfoPrint` 함수는 utilemb.h에서 `EMB_SQL_CHECK`로서 참조됩니다.

Java SQL 오류는 SQLException으로서 드로우되고, 응용프로그램의 catch 블록에서 처리됩니다.

COBOL CHECKERR은 checkerr.cbl라고 하는 외부 프로그램입니다.

REXX CHECKERR은 현재 프로그램의 끝에 있습니다.

이 오류 체크 유틸리티에 대한 소스 코드에 대해서는 137 페이지의 『예제 프로그램에서 GET ERROR MESSAGE 사용』의 내용을 참조하십시오.

C 예: UPDAT.SQC

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlenv.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;  1

int main(int argc, char *argv[])
{

   EXEC SQL BEGIN DECLARE SECTION;  2
      char statement[256];
      char userid[9];
      char passwd[19];
      char jobUpdate[6];
   EXEC SQL END DECLARE SECTION;

   printf( "\nSample C program:  UPDAT \n");

   if (argc == 1)
   {
      EXEC SQL CONNECT TO sample;
  EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3)
   {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;  3
  EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else
   {
      printf ("\nUSAGE: updat [userid passwd]\n\n");
      return 1;
   } /* endif */

   strcpy (jobUpdate, "Clerk");
   EXEC SQL UPDATE staff SET job = :jobUpdate WHERE job = 'Mgr';  4
   EMB_SQL_CHECK("UPDATE STAFF");
   printf ("All 'Mgr' have been demoted to 'Clerk'!\n" );

   strcpy (jobUpdate, "Sales");
   EXEC SQL DELETE FROM staff WHERE job = :jobUpdate;  5
   EMB_SQL_CHECK("DELETE FROM STAFF");
```

```
   printf ("All 'Sales' people have been deleted!\n");

   EXEC SQL INSERT INTO staff
      VALUES (999, 'Testing', 99, :jobUpdate, 0, 0, 0);   6
   EMB_SQL_CHECK("INSERT INTO STAFF");
   printf ("New data has been inserted\n");

   EXEC SQL ROLLBACK;  7
   EMB_SQL_CHECK("ROLLBACK");
   printf( "On second thought -- changes rolled back.\n" );

   EXEC SQL CONNECT RESET;
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}
/* end of program : UPDAT.SQC */
```

Java 예: Updat.sqlj

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;


class Updat
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
    }
    catch (Exception e)
    {   System.out.println ("\n  Error loading DB2 Driver...\n");
      System.out.println (e);
      System.exit(1);
    }
  }


  public static void main(String argv[])
  {   try
    {   System.out.println ("\n  Java Updat Sample");

      String url = "jdbc:db2:sample";       // URL is jdbc:db2:dbname
      Connection con = null;

      // Set the connection                          3
      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception("\nUsage: java Updat [username password]\n");
      }

      // Set the default context
      DefaultContext ctx = new DefaultContext(con);
      DefaultContext.setDefaultContext(ctx);

      // Enable transactions
      con.setAutoCommit(false);

      // UPDATE/DELETE/INSERT
      try
      {   String jobUpdate = null;

        jobUpdate="Clerk";
        #sql {UPDATE staff SET job = :jobUpdate WHERE job = 'Mgr'};   4
        System.out.println("\nAll 'Mgr' have been demoted to 'Clerk'!");

        jobUpdate="Sales";
        #sql {DELETE FROM staff WHERE job = :jobUpdate};
        System.out.println("All 'Sales' people have been deleted!");  5
```

```
        #sql {INSERT INTO staff
          VALUES (999, 'Testing', 99, :jobUpdate, 0, 0, 0)}; 6
        System.out.println("New data has been inserted");
      }
      catch( Exception e )
      {   throw e;
      }
      finally
      {   // Rollback the transaction
        System.out.println("\nRollback the transaction...");
        #sql { ROLLBACK };                                          7
        System.out.println("Rollback done.");
      }
    }
    catch (Exception e)
    {   System.out.println (e);
    }
  }
}
```

COBOL 예: UPDAT.SQB

```
 Identification Division.
 Program-ID. "updat".

 Data Division.
 Working-Storage Section.

     copy "sql.cbl".
     copy "sqlenv.cbl".
     copy "sqlca.cbl".        1

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.          2
 01 statement         pic x(80).
 01 userid            pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
   49 passwd-name     pic x(18).
 01 job-update        pic x(5).
     EXEC SQL END DECLARE SECTION END-EXEC.

* Local variables
 77 errloc          pic x(80).
 77 error-rc        pic s9(9) comp-5.
 77 state-rc        pic s9(9) comp-5.

* Variables for the GET ERROR MESSAGE API
* Use application specific bound instead of BUFFER-SZ
 77 buffer-size     pic s9(4) comp-5 value 1024.
 77 line-width      pic s9(4) comp-5 value 80.
 77 error-buffer    pic x(1024).
 77 state-buffer    pic x(1024).

 Procedure Division.
 Main Section.
     display "Sample COBOL program:  UPDAT".

     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd        3
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

     move "Clerk" to job-update.
     EXEC SQL UPDATE staff SET job=:job-update          4
```

```
         WHERE job='Mgr' END-EXEC.
    move "UPDATE STAFF" to errloc.
    call "checkerr" using SQLCA errloc.

    display "All 'Mgr' have been demoted to 'Clerk'!".

    move "Sales" to job-update.
    EXEC SQL DELETE FROM staff WHERE job=:job-update END-EXEC.    5
    move "DELETE FROM STAFF" to errloc.
    call "checkerr" using SQLCA errloc.

    display "All 'Sales' people have been deleted!".

    EXEC SQL INSERT INTO staff VALUES (999, 'Testing', 99,        6
         :job-update, 0, 0, 0) END-EXEC.
    move "INSERT INTO STAFF" to errloc.
    call "checkerr" using SQLCA errloc.

    display "New data has been inserted".

    EXEC SQL ROLLBACK END-EXEC.                                   7
    move "ROLLBACK" to errloc.
    call "checkerr" using SQLCA errloc.

    DISPLAY "On second thought -- changes rolled back."

    EXEC SQL CONNECT RESET END-EXEC.
    move "CONNECT RESET" to errloc.
    call "checkerr" using SQLCA errloc.

End-Prog.
    stop run.
```

REXX 예: UPDAT.CMD

주: REXX 프로그램에는 정적 SQL을 포함할 수 없습니다. 이 프로그램은 동적 SQL로 작성됩니다.

```
/* REXX program UPDAT.CMD */

parse version rexxType .
parse source platform .

if platform == 'AIX/6000' & rexxType == 'REXXSAA' then
do
  rcy = SysAddFuncPkg("db2rexx")
end
else
do
  if RxFuncQuery('SQLDBS') <> 0 then
    rcy = RxFuncAdd( 'SQLDBS',  'db2ar', 'SQLDBS'  )

  if RxFuncQuery('SQLEXEC') <> 0 then
    rcy = RxFuncAdd( 'SQLEXEC', 'db2ar', 'SQLEXEC' )
end

/* pull in command line arguments */
parse arg userid passwd .

/* check to see if the proper number of arguments have been passed in */
   PARSE ARG dbname userid password .
   if ((dbname = "" ) | ,
       (userid <> "" & password = "") ,
      ) then do
      SAY "USAGE: updat.cmd <dbname> [<userid> <password>]"

        exit -1
   end

   /* connect to database */
   SAY
   SAY 'Connect to' dbname
   IF password= "" THEN
      CALL SQLEXEC 'CONNECT TO' dbname
   ELSE
      CALL SQLEXEC 'CONNECT TO' dbname 'USER' userid 'USING' password

   CALL CHECKERR 'Connect to '
   SAY "Connected"

say 'Sample REXX program: UPDAT.CMD'

jobupdate = "'Clerk'"
st = "UPDATE staff SET job =" jobupdate "WHERE job = 'Mgr'"
call SQLEXEC 'EXECUTE IMMEDIATE :st'  4
call CHECKERR 'UPDATE'
say "All 'Mgr' have been demoted to 'Clerk'!"

jobupdate = "'Sales'"
st = "DELETE FROM staff WHERE job =" jobupdate
call SQLEXEC 'EXECUTE IMMEDIATE :st'  5
```

```
call CHECKERR 'DELETE'
say "All 'Sales' people have been deleted!"

st = "INSERT INTO staff VALUES (999, 'Testing', 99," jobupdate ", 0, 0, 0)"
call SQLEXEC 'EXECUTE IMMEDIATE :st'  6
call CHECKERR 'INSERT'
say 'New data has been inserted'

call SQLEXEC 'ROLLBACK'  7
call CHECKERR 'ROLLBACK'
say 'On second thought...changes rolled back.'

call SQLEXEC 'CONNECT RESET'
call CHECKERR 'CONNECT RESET'


CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else do
    say '--- error report ---'
    say 'ERROR occurred :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /******************************\
    * GET ERROR MESSAGE API called *
    \******************************/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else do
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
return 0
```

진단 조절 및 SQLCA 구조

SQL문 발행 및 데이터베이스 관리 프로그램 API 호출 응용프로그램은 리턴 코드 및 SQLCA 구조를 검토하여 오류 조건을 제대로 점검해야 합니다.

리턴 코드

대부분의 데이터베이스 관리 프로그램 API는 성공시 제로 리턴 코드를 다시 전달합니다. 일반적으로 제로가 아닌 리턴 코드는 2차 오류 처리 메카니즘인 SQLCA 구조가 손상되었음을 나타냅니다. 이런 경우, 호출되는 API가 실행되지 않습니다. SQLCA 구조의 손상은 이 구조에 대해 무효 주소를 전달한 것이 원인일 수 있습니다.

SQLCODE 및 SQLSTATE

오류 정보는 SQLCA 구조의 SQLCODE 및 SQLSTATE 필드에서 리턴되며, 이는 실행가능한 모든 SQL문과 대부분의 데이터베이스 관리 프로그램 API 호출 이후 갱신됩니다.

실행가능 SQL문이 포함된 소스 파일은 `sqlca`라는 이름으로 적어도 하나의 SQLCA 구조를 제공할 수 있습니다. SQLCA 구조는 SQLCA 포함 파일에 정의되어 있습니다. Embedded SQL문은 없지만 호출 데이터베이스 관리 프로그램 API는 있는 소스 파일도 하나 이상의 SQLCA 구조를 제공할 수 있지만 그 이름은 임의적입니다.

응용프로그램이 FIPS 127-2 표준을 따르는 경우, SQLCA 구조를 사용하는 대신 호스트 변수로서 SQLSTATE 및 SQLCODE를 선언할 수 있습니다. 그 방법에 대한 세부사항은, C 또는 C++ 응용프로그램의 경우에는 708 페이지의 『C 및 C++의 SQLSTATE 및 SQLCODE 변수』, COBOL 응용프로그램의 경우에는 785 페이지의 『COBOL에서 SQLSTATE 및 SQLCODE 변수』, FORTRAN 응용프로그램의 경우에는 803 페이지의 『FORTAN에서 SQLSTATE 및 SQLCODE 변수』의 내용을 참조하십시오.

SQLCODE 값 0은 실행이 성공적이었음을 나타냅니다(SQLWARN 경고 조건 가능). 양수 값은 명령문이 성공적으로 실행되었지만 호스트 변수의 절단과 같은 이유로 인해 경고가 있었음을 나타냅니다. 음수 값은 오류 조건이 발생했음을 나타냅니다.

추가 필드인 SQLSTATE에는 다른 IBM 데이터베이스 제품 및 SQL92형 데이터베이스 관리 프로그램을 통해 일관된 표준화된 오류 코드가 들어 있습니다. 실질적으로, 대부분의 데이터베이스 관리 프로그램을 통해 SQLSTATE가 일반적이므로, 이식성을 생각할 때 SQLSTATE를 사용해야 합니다.

SQLCODE가 제로인 경우에도 SQLWARN 필드에는 경고 표시기 배열이 포함되어 있습니다. 다른 모든 요소가 공백인 경우, SQLWARN 배열의 첫번째 요소인 SQLWARN0에는 공백이 포함되어 있습니다. 적어도 다른 한 요소에 경고 문자가 있을 경우 SQLWARN0에는 W가 포함되어 있습니다.

SQLCA 구조에 대한 세부사항은 *Administrative API Reference*>, SQLCODE 및 SQLSTATE 오류 조건 목록은 *메시지 참조서*의 내용을 참조하십시오.

주: 다양한 IBM RDBMS 서버에 액세스하는 응용프로그램을 개발하려면 다음을 수행해야 합니다.

- 가능한 경우, 사용자 응용프로그램이 SQLCODE가 아닌 SQLSTATE를 점검하도록 하십시오.
- 응용프로그램이 DB2 Connect를 사용할 경우, 닮지 않은(unlike) 데이터베이스 사이에 SQLCODE 변환을 맵하려면 DB2 Connect가 제공하는 맵핑 기능을 사용해 보십시오.

SQLCA 구조에서의 토큰 절단

토큰이 SQLCA 구조에서 절단될 수 있으므로 토큰 정보를 진단용으로 사용해서는 안됩니다. 테이블 및 컬럼 이름을 최고 128바이트 길이로 정의할 수 있지만 SQLCA 토큰은 17바이트와 절단 종료자(>) 길이를 합한 길이로 절단됩니다. 응용프로그램 논리는 `sqlerrmc` 필드의 실제 길이에 의존해서는 안됩니다. 토큰 절단에 대한 논의 및 SQLCA 구조에 대한 설명은 *SQL 참조서*의 내용을 참조하십시오.

WHENEVER문 사용시 오류 처리

오류나 경고가 있거나, 실행 중에 발견되는 행이 없는 경우 WHENEVER문은, 사전 처리 컴파일러가 지정된 레이블로 가도록 응용프로그램을 지정하는 소스 코드를 생성하도록 합니다. WHENEVER문은 다른 WHENEVER문이 상황을 변경할 때까지, 이후에 오는 모든 실행가능 SQL문에 영향을 미칩니다.

WHENEVER문은 다음과 같은 세 가지 기본 형식을 가지고 있습니다.

```
EXEC SQL WHENEVER SQLERROR   action
EXEC SQL WHENEVER SQLWARNING action
EXEC SQL WHENEVER NOT FOUND  action
```

위 명령문에서:

SQLERROR
: SQLCODE < 0인 조건을 식별합니다.

SQLWARNING
: SQLWARN(0) = W 또는 SQLCODE > 0이지만 100과 같지는 않은 조건을 식별합니다.

NOT FOUND
: SQLCODE = 100인 조건을 식별합니다.

어떤 경우이든, *조치*는 다음과 같습니다.

CONTINUE
: 응용프로그램의 다음 단계를 계속 진행합니다.

GO TO *label*
: GO TO 다음에 지정된 레이블 바로 뒤에 오는 명령문으로 갑니다. (GO TO는 두 단어로도 쓰고, 한 단어 GOTO로도 쓸 수 있습니다.)

WHENEVER문을 사용하지 않는 경우, 실행 중 오류, 경고 또는 예외가 발생할 경우 처리를 계속하는 것이 기본 조치입니다.

WHENEVER문은 영향을 주려는 SQL문 앞에 와야 합니다. 그렇지 않으면, 사전 처리 컴파일러는 실행가능한 SQL문에 대해 추가 오류 처리 코드를 생성해야 한다는 것을 인식하지 못합니다. 언제라도 세 가지 기본 형식을 조합하여 사용할 수

있습니다. 세 형식의 선언 순서는 중요하지 않습니다. 무한 루핑을 피하려면 SQL 문이 핸들러 내부에서 실행되기 전에 WHENEVER 조절을 실행 취소해야 합니다. WHENEVER SQLERROR CONTINUE문을 사용하여 이를 수행할 수 있습니다.

WHENEVER문에 대한 전체 설명은 *SQL 참조서*에서 자세한 내용을 참조하십시오.

예외, 신호, 인터럽트 핸들러 고려사항

예외, 신호 또는 인터럽트 핸들러는 예외, 신호 또는 인터럽트가 발생할 때 제어를 하는 루틴입니다. 적용가능한 핸들러 유형은 다음에서와 같이 사용자의 운영 환경에 따라 결정됩니다.

Windows 32비트 운영 체제
: Ctrl-C 또는 Ctrl-Break를 누르면 인터럽트가 발생합니다.

OS/2
: Ctrl-C 또는 Ctrl-Break를 누르면 운영 체제 예외가 발생합니다.

UNIX
: 일반적으로 Ctrl-C를 누르면 SIGINT 인터럽트 신호가 생성됩니다. 키보드를 다시 정의하는 것은 쉬우므로, 사용자 머신에 서로 다른 키 절차로 인해 SIGINT가 생성될 수 있습니다.

위 목록에 없는 운영 체제의 경우는 *응용프로그램 빌드 안내서*의 내용을 참조하십시오.

예외, 신호 및 인터럽트 핸들러에 (COMMIT 또는 ROLLBACK이 아닌) SQL문을 두지 마십시오. 이러한 종류의 오류 조건에서는 일반적으로 데이터가 불일치하는 위험을 피하기 위해 ROLLBACK을 수행하려고 합니다.

예외/신호/인터럽트 핸들러에서 COMMIT 및 ROLLBACK을 코드화할 때 주의를 실행해야 합니다. 이 명령문들 중 하나를 호출할 경우, 현재 SQL문이 수행중일 때에는 현재 SQL문이 완료될 때까지 COMMIT이나 ROLLBACK은 실행되지 않습니다. 이는 Ctrl-C 핸들러에서 원하는 작동은 아닙니다.

솔루션은 ROLLBACK을 발행하기 전에 INTERRUPT API(sqleintr/sqlgintr)를 호출하는 것입니다. 이는 (응용프로그램이 현재 SQL 조회를 실행하고 있을 경

우) 현재 SQL 조회를 인터럽트하고 ROLLBACK이 즉시 시작되도록 합니다. ROLLBACK이 아닌 COMMIT를 수행하려고 할 경우 현재 명령을 인터럽트하지 않습니다.

APPC를 사용하여 원격 데이터베이스 서버(DB2 Connect를 사용하는 호스트 데이터베이스 시스템 또는 AIX용 DB2)에 액세스할 때 응용프로그램은 SIGUSR1 신호를 받을 수 있습니다. 이 신호는 복구할 수 없는 오류가 발생하여 SNA 연결이 중지될 때 SNA Services/6000에 의해 생성됩니다. 사용자 응용프로그램에 신호 핸들러를 설치하여 SIGUSR1을 처리해야 할 것입니다.

다양한 핸들러 고려사항에 대한 세부사항은 플랫폼 문서를 참조하십시오.

exit 목록 루틴 고려사항

exit 목록 루틴에서 SQL 또는 DB2 API 호출을 사용하지 마십시오. exit 루틴에 있는 데이터베이스로부터 연결해제할 수 없습니다.

예제 프로그램에서 GET ERROR MESSAGE 사용

『C 예: UTILAPI.C』 및 141 페이지의 『COBOL 예: CHECKERR.CBL』에 표시된 코드 클립들은, 전달된 SQLCA와 관련된 정보를 얻기 위한 GET ERROR MESSAGE API의 사용 예를 보여줍니다.

이러한 예의 구축에 대해서는 README 파일이나 이 샘플 프로그램의 헤더 절을 참조하십시오.

C 예: UTILAPI.C

```
#include <stdio.h>
#include <stdlib.h>
#include <sql.h>
#include <sqlenv.h>
#include <sqlda.h>
#include <sqlca.h>
#include <string.h>
#include <ctype.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

/*#################################################################
**    1. SQL_CHECK section
**
**        1.1 - SqlInfoPrint - prints on the screen everything that
**                             goes unexpected.
```

```
**        1.2 - TransRollback - rolls back the transaction
##################################################################*/

/******************************************************************************
**        1.1 - SqlInfoPrint - prints on the screen everything that
**                             goes unexpected.
******************************************************************************/
int SqlInfoPrint( char *         appMsg,
                  struct sqlca * pSqlca,
                  int            line,
                  char *         file )
{   int   rc = 0;

    char  sqlInfo[1024];
    char  sqlInfoToken[1024];

    char  sqlstateMsg[1024];
    char  errorMsg[1024];


    if (pSqlca->sqlcode != 0 && pSqlca->sqlcode != 100)
    {   strcpy(sqlInfo, "");

        if( pSqlca->sqlcode < 0)
        {   sprintf( sqlInfoToken, "\n---- error report
----\n");
            strcat( sqlInfo, sqlInfoToken);
        }
        else
        {   sprintf( sqlInfoToken, "\n---- warning report
----\n");
            strcat( sqlInfo, sqlInfoToken);
        } /* endif */

        sprintf( sqlInfoToken, "  app. message      = %s\n", appMsg);
        strcat( sqlInfo, sqlInfoToken);
        sprintf( sqlInfoToken, "  line              = %d\n", line);
        strcat( sqlInfo, sqlInfoToken);
        sprintf( sqlInfoToken, "  file              = %s\n", file);
        strcat( sqlInfo, sqlInfoToken);
        sprintf( sqlInfoToken, "  SQLCODE           = %ld\n", pSqlca->sqlcode);
        strcat( sqlInfo, sqlInfoToken);

        /* get error message */
        rc = sqlaintp( errorMsg, 1024, 80, pSqlca);
        /* return code is the length of the errorMsg string */
        if( rc > 0)
        {   sprintf( sqlInfoToken, "%s\n", errorMsg);
            strcat( sqlInfo, sqlInfoToken);
        }

        /* get SQLSTATE message */
        rc = sqlogstt( sqlstateMsg, 1024, 80, pSqlca->sqlstate);
        if (rc == 0)
        {   sprintf( sqlInfoToken, "%s\n", sqlstateMsg);
            strcat( sqlInfo, sqlInfoToken);
        }


        if( pSqlca->sqlcode < 0)
        {   sprintf( sqlInfoToken, "--- end error report ---\n");
            strcat( sqlInfo, sqlInfoToken);
```

```
            printf("%s", sqlInfo);
            return 1;
        }
        else
        {   sprintf( sqlInfoToken, "--- end warning report ---\n");
            strcat( sqlInfo, sqlInfoToken);

            printf("%s", sqlInfo);
            return 0;
        } /* endif */
    } /* endif */

    return 0;
}

/******************************************************************************
**       1.2 - TransRollback - rolls back the transaction
******************************************************************************/
void TransRollback( )
{   int         rc = 0;

    /* rollback the transaction */
    printf( "\nRolling back the transaction ...\n") ;
    EXEC SQL ROLLBACK;
    rc = SqlInfoPrint( "ROLLBACK", &sqlca, __LINE__, __FILE__);
    if( rc == 0)
    {   printf( "The transaction was rolled back.\n") ;
    }

}
```

Java 예: SQLException 캐치

JDBC 및 SQLJ 응용프로그램은 SQL 처리 중에 오류가 발생할 때 SQLException을 드로우(throw)합니다. 응용프로그램은 다음 코드를 이용하여 SQLException을 포착 및 표시할 수 있습니다.

```
try {
   Statement stmt = connection.createStatement();
   int rowsDeleted = stmt.executeUpdate(
      "DELETE FROM employee WHERE empno = '000010'");
   System.out.println( rowsDeleted + " rows were deleted");
}

catch (SQLException sqle) {
   System.out.println(sqle);
}
```

SQLExceptions을 조절하는 방법에 대해서는 714 페이지의 『Java에 있는 SQLSTATE 및 SQLCODE 값』의 내용을 참조하십시오.

COBOL 예: CHECKERR.CBL

```
 Identification Division.
 Program-ID. "checkerr".

 Data Division.
 Working-Storage Section.

 copy "sql.cbl".

* Local variables
 77 error-rc        pic s9(9) comp-5.
 77 state-rc        pic s9(9) comp-5.

* Variables for the GET ERROR MESSAGE API
* Use application specific bound instead of BUFFER-SZ
* 77 buffer-size     pic s9(4) comp-5 value BUFFER-SZ.
* 77 error-buffer    pic x(BUFFER-SZ).
* 77 state-buffer    pic x(BUFFER-SZ).
 77 buffer-size     pic s9(4) comp-5 value 1024.
 77 line-width      pic s9(4) comp-5 value 80.
 77 error-buffer    pic x(1024).
 77 state-buffer    pic x(1024).

 Linkage Section.
 copy "sqlca.cbl" replacing ==VALUE "SQLCA   "== by == ==
                            ==VALUE 136==        by == ==.
 01 errloc          pic x(80).

 Procedure Division using sqlca errloc.
 Checkerr Section.
     if SQLCODE equal 0
        go to End-Checkerr.

     display "--- error report ---".
     display "ERROR occurred : ", errloc.
     display "SQLCODE : ", SQLCODE.

************************;*******
* GET ERROR MESSAGE API called *
************************;*******
     call "sqlgintp" using
                            by value     buffer-size
                            by value     line-width
                            by reference sqlca
                            by reference error-buffer
                      returning error-rc.

************************
* GET SQLSTATE MESSAGE *
************************
     call "sqlggstt" using
                            by value     buffer-size
                            by value     line-width
                            by reference sqlstate
                            by reference state-buffer
                      returning state-rc.

     if error-rc is greater than 0
```

```
         display error-buffer.

      if state-rc is greater than 0
         display state-buffer.

      if state-rc is less than 0
         display "return code from GET SQLSTATE =" state-rc.

      if SQLCODE is less than 0
         display "--- end error report ---"
         go to End-Prog.

      display "--- end error report ---"
      display "CONTINUING PROGRAM WITH WARNINGS!".
  End-Checkerr. exit program.
  End-Prog. stop run.
```

REXX 예: CHECKERR 프로시듀어

```
parse version rexxType .
parse source platform .

if platform == 'AIX/6000' & rexxType == 'REXXSAA' then
do
  rcy = SysAddFuncPkg("db2rexx")
end
else
do
  if RxFuncQuery('SQLDBS') <> 0 then
    rcy = RxFuncAdd( 'SQLDBS',  'db2ar', 'SQLDBS'  )

  if RxFuncQuery('SQLEXEC') <> 0 then
    rcy = RxFuncAdd( 'SQLEXEC', 'db2ar', 'SQLEXEC' )
end

⋮

call CHECKERR 'INSERT'

⋮

CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else do
    say '--- error report ---'
    say 'ERROR occurred :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /******************************\
    * GET ERROR MESSAGE API called *
    \******************************/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else do
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
```

```
return 0

/* this variable (SYSTEM) must be user defined */
SYSTEM = AIX
if SYSTEM = OS2 then do
  if RxFuncQuery('SQLDBS') <> 0 then
    rcy = RxFuncAdd( 'SQLDBS',  'DB2AR', 'SQLDBS'  )

  if RxFuncQuery('SQLEXEC') <> 0 then
    rcy = RxFuncAdd( 'SQLEXEC', 'DB2AR', 'SQLEXEC' )
end

if SYSTEM = AIX then
  rcy = SysAddFuncPkg("db2rexx")

⋮

call CHECKERR 'INSERT'

⋮

CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else do
    say '--- error report ---'
    say 'ERROR occurred :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /*****************************\
    * GET ERROR MESSAGE API called *
    \*****************************/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else do
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
return 0
```

제5장 동적 SQL 프로그램 작성

동적 SQL 사용 이유 145
동적 SQL 지원 명령문 146
동적 SQL과 정적 SQL 비교 147
PREPARE, DESCRIBE, FETCH 및 SQLDA 사용 150
커서 선언 및 사용 151
예: 동적 SQL 프로그램 152
동적 프로그램 작업 방법 152
C 예: DYNAMIC.SQC 154
Java 예: Dynamic.java. 156
COBOL 예: DYNAMIC.SQB 158
REXX 예: DYNAMIC.CMD 160
SQLDA 선언 162
최소 SQLDA 구조를 사용하는 명령문 준비 163
충분한 SQLVAR 항목으로 SQLDA 할당 165
SELECT문 설명 166
행 보유를 위한 저장영역 취득 166
커서 처리 167
SQLDA 구조 할당 167
SQLDA 구조를 사용한 데이터 전달 . . 171
대화식 SQL문 처리 172
명령문 유형 결정 172
Varying-List SELECT문 173
일반 사용자로부터 SQL 요청 저장 . . . 174
예: ADHOC 프로그램 174
ADHOC 프로그램 작업 방법. 174
C 예: ADHOC.SQC 177
동적 SQL로의 변수 입력 180
매개변수 표시문자 사용. 180
예: VARINP 프로그램 181
VARINP 프로그램 작업 방법 181
C 예: VARINP.SQC 183
Java 예: Varinp.java 185
COBOL 예: VARINP.SQB 187
DB2 Call Level Interface(CLI) 189
Embedded SQL 및 DB2 CLI 비교 . . 189
DB2 CLI 사용시 이점 191
Embedded SQL 또는 DB2 CLI에 대한 결정 193

동적 SQL 사용 이유

다음과 같은 경우 동적 SQL을 사용해야 할 것입니다.

- 응용프로그램 실행 중에 SQL문 전체 또는 일부를 생성해야 하는 경우.
- SQL문에 의해 참조되는 오브젝트가 사전 처리 컴파일될 때 없는 경우.
- 명령문이 항상 현재 데이터베이스 통계에 기초하여 최적의 액세스 경로를 사용하기를 원하는 경우.
- 명령문의 컴파일 환경을 수정하려는 경우, 즉 특수 레지스터와의 실험을 원하는 경우.

© Copyright IBM Corp. 1993, 2000

동적 SQL 지원 명령문

동적 SQL 지원 명령문은 인수로서 문자열 호스트 변수 및 명령문 이름을 허용합니다. 호스트 변수에는 텍스트 형식으로 동적으로 처리할 SQL문이 포함되어 있습니다. 응용프로그램이 사전 처리 컴파일될 때 명령문 텍스트는 처리되지 않습니다. 사실 명령문 텍스트는 응용프로그램이 사전 처리 컴파일될 때 존재할 필요가 없습니다. 대신, SQL문은 사전 처리 컴파일용 호스트 변수로서 취급되고, 이 변수는 응용프로그램 실행 중에 참조됩니다. 이러한 SQL문을 동적 SQL이라고 합니다.

SQL 텍스트가 포함된 호스트 변수를 실행가능 형식으로 변환하고, 명령문 이름을 참조하여 이에 대한 작업을 할 때 동적 SQL 지원 명령문이 필요합니다. 이 명령문은 다음과 같습니다.

EXECUTE IMMEDIATE
: 호스트 변수를 사용하지 않는 명령문을 준비 및 실행합니다. 응용프로그램의 모든 EXECUTE IMMEDIATE문은 런타임시 같은 위치에 캐쉬되므로, 최종 명령문만 알려집니다. 이 명령문을 PREPARE문 및 EXECUTE문의 대안으로서 사용하십시오.

PREPARE
: SQL문의 문자열 형식을 명령문의 실행가능 형식으로 전환하고, 명령문 이름을 지정하며, SQLDA 구조에 명령문에 대한 정보를 선택적으로 둡니다.

EXECUTE
: 이전에 준비된 SQL문을 실행합니다. 명령문은 연결 내에서 반복적으로 실행될 수 있습니다.

DESCRIBE
: 준비된 명령문에 대한 정보를 SQLDA에 둡니다.

응용프로그램은 대부분의 SQL문을 동적으로 실행할 수 있습니다. 지원되는 SQL문의 전체 목록은 829 페이지의 표37의 내용을 참조하십시오.

주: 동적 SQL문 내용 앞에는 정적 SQL문과 같은 구문이 나오지만, 다음과 같은 예외가 있습니다.

- 주석은 허용되지 않습니다.
- 명령문은 EXEC SQL로 시작할 수 없습니다.

- 명령문은 명령문 종료자로 종료될 수 없습니다. 이에 대한 예외로는 세미콜론(;)이 포함될 수 있는 CREATE TRIGGER문입니다.

동적 SQL과 정적 SQL 비교

정적 SQL과 동적 SQL 중에서 어느 것을 사용해야 할 것인지는, 성능을 고려해야 하는 프로그래머에게 있어서는 중요한 문제입니다. 그 대답은 물론, 전적으로 상황에 따라 다르다는 것입니다. 정적 SQL을 사용할 것인지 동적 SQL을 사용할 것인지 선택할 때 도움말을 보려면 표6의 내용을 참조하십시오. 정적 SQL을 명시하는 보안이나 동적 SQL을 명시하는 환경(예. DB2 CLI 또는 CLP 사용 환경)과 같은 고려사항들이 있을 수 있습니다.

의사결정을 할 때 특정 상황에서 정적 SQL을 선택할 것인지 동적 SQL을 선택할 때 다음 사항을 고려하십시오. 다음 테이블에서 'either'는 정적 SQL이나 동적 SQL, 어느 것을 선택해도 두드러진 장점이 없음을 의미합니다. 이는 일반적인 권고사항일 뿐입니다. 특정 응용프로그램, 사용 목적, 작업 환경 등이 실제 선택사항을 명시합니다. 의심될 경우, 사용자의 명령문을 정적 SQL로 프로토타입한 후 동적 SQL로 프로토타입하고, 그 차이점을 비교해 보는 것이 가장 좋습니다.

표 6. 정적 SQL과 동적 SQL 비교

고려사항	최적 선택사항
SQL문 수행 시간:	
• 2초 이하	• 정적
• 2-10초	• 두개 중 하나
• 10초 이상	• 동적
데이터 단일성	
• 단일 데이터 분산	• 정적
• 약간의 비단일성	• 두개 중 하나
• 비단일성 분산	• 동적
범위(<,>,BETWEEN,LIKE) 술어	
• 드묾	• 정적
• 간헐적	• 두개 중 하나
• 빈번	• 동적
반복 실행	
• 여러 번 수행(10번 이상)	• 두개 중 하나
• 자주 수행(10번 이하)	• 두개 중 하나
• 한 번 수행	• 정적

표 6. 정적 SQL과 동적 SQL 비교 (계속)

고려사항	최적 선택사항
조회 특성	
• 무작위	• 동적
• 영구	• 두개 중 하나
런타임 환경 (DML/DDL)	
• 트랜잭션 처리 (DML 만)	• 두개 중 하나
• 혼합(DML 및 DDL - DDL은 패키지에 영향을 줍니다.)	• 동적
• 혼합(DML 및 DDL - DDL은 패키지에 영향을 줍니다.)	• 두개 중 하나
RUNSTATS 빈도	
• 드묾	• 정적
• 정기적	• 두개 중 하나
• 빈번	• 동적

일반적으로 동적 SQL을 사용하는 응용프로그램은 사용하기 전에 SQL문을 컴파일해야 할 필요에 의해 SQL문 하나당 더 높은 시동(또는 시작) 비용이 있습니다. 일단 컴파일되면 정적 SQL에 비교하여 동적 SQL 실행 시간은 같아야 하며, 최적화 알고리즘에 의해 선택되는 계획에 더 잘 액세스할 수 있음으로 인해 속도가 더 빨라질 수 있습니다. 동적 명령문이 실행될 때마다 초기 컴파일 비용은 하나의 인수 이하가 됩니다. 다중 사용자가 동일한 명령문을 가진 동일한 동적 응용프로그램을 수행할 경우, 명령문을 실행하는 첫번째 응용프로그램만 명령문 컴파일 비용을 실현합니다.

혼합 DML 및 DDL 환경에서 동적 SQL문에 대한 컴파일 비용은 응용프로그램이 수행되는 동안 시스템에 의해 명령문이 암시적으로 재컴파일될 때 달라질 수 있습니다. 혼합 환경에서 정적 및 동적 SQL 사이의 선택항목은 패키지가 무효화되는 빈도의 요인이 되어야 합니다. DDL이 패키지를 무효화할 경우, 동적 SQL은 실행되는 조회가 다음에 사용될 때 재컴파일될수록 효율적이 됩니다. 다른 동적 SQL은 재컴파일되지 않습니다. 정적 SQL의 경우전체 패키지는 일단 무효화되면 재바인드됩니다.

이제 사용자의 특정 응용프로그램에 위 특성들이 혼합되어 있고, 이 특성 중 일부는 정적 SQL을 사용하고, 다른 특성들은 동적 SQL을 사용해야 한다고 가정하십

시오. 이런 경우, 확연한 결정사항은 없고, 가장 많은 경험이 있어 가장 편안하게 느껴지는 방법을 사용해야 합니다. 위 테이블에서 유의사항은 중요도 순으로 나열됩니다.

주: 정적 및 동적 SQL은 각각 DB2 최적화 알고리즘과 차이를 보이는 두 가지 유형을 가집니다. 이들은 다음과 같습니다.

1. 호스트 변수가 없는 정적 SQL

   다음과 같은 경우 잘 이루어지지 않습니다.

   - *초기화 코드*
   - *초심자 훈련 예*

   런타임 성능 오버헤드가 없지만 DB2 최적화 알고리즘의 기능이 완전히 실현될 수 있다는 점에서 성능상 실제로 가장 좋은 조합이라고 할 수 있습니다.

2. 호스트 변수가 포함된 정적 SQL

   이것은 기존의 DB2 응용프로그램의 *legacy* 스타일입니다. 명령문 컴파일 중에 취득한 카탈로그 잠금 및 PREPARE의 런타임 오버헤드를 방지합니다. 최적화 알고리즘의 전기능이 전체 SQL문을 알 수는 없으므로 장비화할 수는 없습니다. 단일하지 않은 데이터 분산에 있어 특정 문제가 있습니다.

3. 매개변수 표시문자가 없는 동적 SQL

   이것은 (CLP와 같은) 임의의 조회 인터페이스에 대한 일반 스타일이며, SQL의 최적화 알고리즘 선호 스타일입니다. 일반적으로, 복합 조회의 경우 PREPARE문의 오버헤드는 실행 시간이 향상되었으므로 가치가 있습니다. 매개변수 표시문자에 대한 세부사항은 180 페이지의 『매개변수 표시문자 사용』의 내용을 참조하십시오.

4. 매개변수 표시문자가 포함된 동적 SQL

   CLI 응용프로그램에 대해 가장 일반적인 SQL 유형입니다. 주요 장점은 매개변수 표시문자의 존재로 인해 select 또는 insert문의 반복된 실행을 통해 PREPARE 비용이 상환된다는 점입니다. 이 상환은 모든 동적 SQL 응용프로그램에 적용됩니다. 호스트 변수를 가진 정적 SQL과 마찬가지로, 전체 정보를 사용할 수 없음으로 인해 DB2 최적화 알고리즘의 일부가 작동되지 않게 됩니다. 가장 효율적인 옵션은 *호스트 변수와 함께 정적 SQL*을 사용하거나 *매개변수 표시문자 없이 동적 SQL*을 사용하는 것입니다.

PREPARE, DESCRIBE, FETCH 및 SQLDA 사용

정적 SQL을 이용하여 Embedded SQL문에서 사용되는 호스트 변수는 응용프로그램이 컴파일될 때 알려집니다. 동적 SQL을 이용하여 Embedded SQL문 및 결과적으로 호스트 변수는 응용프로그램이 런타임될 때까지 알려지지 않습니다. 따라서, 동적 SQL 응용프로그램의 경우 응용프로그램에서 사용되는 호스트 변수 목록을 다루어야 합니다. (PREPARE를 사용하여) 준비된 SELECT문에 대한 호스트 변수 정보를 얻고, 이 정보를 SQL 설명자 영역(SQLDA)에 저장하기 위해 DESCRIBE문을 사용할 수 있습니다.

주: Java 응용프로그램은 SQLDA 구조를 사용하지 않으므로, PREPARE문 또는 DESCRIBE문을 사용하지 않습니다. JDBC 응용프로그램에서는 `PreparedStatement` 오브젝트 및 `executeQuery()` 메소드를 사용하여 호스트 언어 커서와 같은 `ResultSet` 오브젝트를 생성할 수 있습니다. SQLJ 응용프로그램에서는 `CursorByPos` 또는 `CursorByName` 커서와 함께 SQLJ `iterator` 오브젝트를 선언하여 FETCH문으로부터 데이터를 리턴할 수 있습니다.

DESCRIBE문이 사용자 응용프로그램에서 실행될 때 데이터베이스 관리 프로그램은 호스트 변수를 SQLDA에 정의합니다. 호스트 변수가 SQLDA에 정의되면, 커서를 사용하여 값을 호스트 변수에 지정하기 위해 FETCH문을 사용할 수 있습니다.

PREPARE, DESCRIBE 및 FETCH문에 대한 자세한 정보 및 SQLDA에 대한 설명은 *SQL 참조서*의 내용을 참조하십시오.

SQLDA를 사용하지 않고 PREPARE, DESCRIBE 및 FETCH문을 사용하는 간단한 동적 SQL 프로그램의 예는 152 페이지의 『예: 동적 SQL 프로그램』의 내용을 참조하십시오. 대화식 SQL문을 처리하기 위해 PREPARE, DESCRIBE 및 FETCH문과 SQLDA를 사용하는 동적 SQL 프로그램의 예는 174 페이지의 『예: ADHOC 프로그램』의 내용을 참조하십시오.

커서 선언 및 사용

커서의 동적 처리는 정적 SQL을 사용하여 처리하는 것과 거의 동일합니다. 커서가 선언될 때 조회와 연관되어 있습니다.

정적 SQL의 경우 조회는 96 페이지의 『커서문 정의』에서와 같이, 텍스트 형식으로 된 SELECT문입니다.

동적 SQL의 경우, 조회는 PREPARE문에 지정된 명령문 이름과 연관되어 있습니다. 참조된 호스트 변수는 매개변수 표시문자에 의해 표시됩니다. 표7에는 동적 SELECT문과 연관된 DECLARE문이 표시되어 있습니다.

표 7. 동적 SELECT와 연관된 명령문 선언

언어	예제 소스 코드

C/C++	`strcpy( prep_string, "SELECT tabname FROM syscat.tables"`
	`                      "WHERE tabschema = ?" );`
	`EXEC SQL PREPARE s1 FROM :prep_string;`
	`EXEC SQL DECLARE c1 CURSOR FOR s1;`
	`EXEC SQL OPEN c1 USING :host_var;`

Java (JDBC)	`PreparedStatement prep_string = ("SELECT tabname FROM syscat.tables`
	`                      WHERE tabschema = ?" );`
	`prep_string.setCursor("c1");`
	`prep_string.setString(1, host_var);`
	`ResultSet rs = prep_string.executeQuery();`

COBOL	`MOVE "SELECT TABNAME FROM SYSCAT.TABLES WHERE TABSCHEMA = ?"`
	`     TO PREP-STRING.`
	`EXEC SQL PREPARE S1 FROM :PREP-STRING END-EXEC.`
	`EXEC SQL DECLARE C1 CURSOR FOR S1 END-EXEC.`
	`EXEC SQL OPEN C1 USING :host-var END-EXEC.`

FORTRAN	`prep_string = 'SELECT tabname FROM syscat.tables WHERE tabschema = ?'`
	`EXEC SQL PREPARE s1 FROM :prep_string`
	`EXEC SQL DECLARE c1 CURSOR FOR s1`
	`EXEC SQL OPEN c1 USING :host_var`

정적 커서와 동적 커서 사이의 주요 차이점은 정적 커서는 사전 처리 컴파일될 때 준비되고, 동적 커서는 수행될 때 준비됩니다. 또한, 조회에서 참조되는 호스트 변수는 매개변수 표시문자에 의해 표시되고, 매개변수 표시문자는 커서가 열릴 때 런타임 호스트 변수에 의해 대체됩니다.

커서 사용 방법에 대한 자세한 내용은 다음 절을 참조하십시오.

- 95 페이지의 『커서를 이용한 다중 행 선택』
- 99 페이지의 『예: 커서 프로그램』
- 819 페이지의 『REXX에서 커서 사용』

예: 동적 SQL 프로그램

이 샘플 프로그램은 동적 SQL문에 기초한 커서의 처리를 표시합니다. 이 프로그램은 이름 컬럼에 STAFF 값을 가진 테이블을 제외하고, SYSCAT.TABLES에 모든 테이블을 나열합니다. 샘플은 다음과 같은 프로그래밍 언어로 사용할 수 있습니다:

C dynamic.sqc

Java Dynamic.java

COBOL dynamic.sqb

REXX dynamic.cmd

동적 프로그램 작업 방법

1. **호스트 변수 정의.** 이 절에서는 3개의 호스트 변수의 선언이 있습니다.

   table_name
   FETCH문이 실행되는 동안 리턴되는 데이터를 보유하는 데 사용됩니다.

   st 동적 SQL문을 텍스트 형식으로 보유하는 데 사용됩니다.

   parm_var
   st에서 매개변수 표시문자를 대체할 데이터 값을 제공합니다.

2. **명령문 준비.** ('?'에 의해 표시되는) 하나의 매개변수 표시문자를 가진 SQL문이 호스트 변수로 복사됩니다. 이 호스트 변수는 검증을 위해 PREPARE문으로 전달됩니다. PREPARE문은 SQL 텍스트를 분석하고 사전 처리 컴파일러 또는 바인더와 같은 방식으로 패키지에 대한 액세스 절을 준비하는데, 이는 선행 처리 중이 아닌 수행 중에 이루어집니다.

3. **커서 선언.** DECLARE문은 커서를 동적으로 준비된 SQL문에 연결합니다. 준비된 SQL문이 SELECT문인 경우, 결과 테이블로부터 행을 검색하는 데 커서가 필수적입니다.

4. **커서 열기.** OPEN문은 이전에 선언된 커서를 초기설정하여 결과 테이블의 첫 번째 행 앞을 가리킵니다. USING절은 호스트 변수를 지정하여 준비된 SQL 문에 있는 매개변수 표시문자를 대체합니다. 호스트 변수 길이 및 데이터 유형은 연관된 컬럼 유형 및 길이와 호환되어야 합니다.

5. **데이터 검색.** FETCH문은 결과 테이블로부터 table_name 호스트 변수로 NAME 컬럼을 이동합니다. 호스트 변수는 프로그램이 다른 행을 페치하기 위해 루프백하기 전에 인쇄됩니다.

6. **커서 닫기.** CLOSE문은 커서를 닫고 이와 연관된 자원을 해제합니다.

CHECKERR 매크로/함수는 오류 체크 유틸리티입니다. 이 오류 체크 유틸리티의 위치는 사용된 프로그래밍 언어에 따라 달라질 수 있습니다.

C
: DB2 API를 호출하는 C 프로그램에 대해서는 utilapi.c에서 sqlInfoPrint 함수는 utilapi.h에서 API_SQL_CHECK로서 참조됩니다. C embedded SQL 프로그램에 대해서는 utilemb.sqc에서 sqlInfoPrint 함수는 utilemb.h에서 EMB_SQL_CHECK로서 참조됩니다.

Java
: SQL 오류는 SQLException으로서 드로우되고, 응용프로그램의 catch 블록에서 처리됩니다.

COBOL
: CHECKERR은 checkerr.cbl라고 하는 외부 프로그램입니다.

REXX
: CHECKERR은 현재 프로그램의 끝에 있습니다.

이 오류 체크 유틸리티에 대한 소스 코드에 대해서는 137 페이지의『예제 프로그램에서 GET ERROR MESSAGE 사용』의 내용을 참조하십시오.

C 예: DYNAMIC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION;
      char  table_name[19];
      char  st[80];   1
      char  parm_var[19];
      char userid[9];
      char passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: DYNAMIC\n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else {
      printf ("\nUSAGE: dynamic [userid passwd]\n\n");
      return 1;
   } /* endif */

   strcpy( st, "SELECT tabname FROM syscat.tables" );
   strcat( st, " WHERE tabname <> ?" );
   EXEC SQL PREPARE s1 FROM :st;   2
   EMB_SQL_CHECK("PREPARE");

   EXEC SQL DECLARE c1 CURSOR FOR s1;   3

   strcpy( parm_var, "STAFF" );
   EXEC SQL OPEN c1 USING :parm_var;   4
   EMB_SQL_CHECK("OPEN");
   do {
      EXEC SQL FETCH c1 INTO :table_name;   5
```

```
      if (SQLCODE != 0) break;

      printf( "Table = %s\n", table_name );
   } while ( 1 );

   EXEC SQL CLOSE c1;   6
   EMB_SQL_CHECK("CLOSE");

   EXEC SQL COMMIT;
   EMB_SQL_CHECK("COMMIT");

   EXEC SQL CONNECT RESET;
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}
/* end of program : DYNAMIC.SQC */
```

Java 예: Dynamic.java

```
import java.sql.*;

class Dynamic
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
    }
    catch (Exception e)
    {   System.out.println ("\n  Error loading DB2 Driver...\n");
      System.out.println (e);
      System.exit(1);
    }
  }

  public static void main(String argv[])
  {   try
    {   System.out.println ("  Java Dynamic Sample");
      // Connect to Sample database

      Connection con = null;
      // URL is jdbc:db2:dbname
      String url = "jdbc:db2:sample";

      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception("\nUsage: java Dynamic [username password]\n");
      }

      // Enable transactions
      con.setAutoCommit(false);

      // Perform dynamic SQL SELECT using JDBC
      try
      {   PreparedStatement pstmt1 = con.prepareStatement(
          "SELECT tabname FROM syscat.tables " +
          "WHERE tabname <> ? " +
          "ORDER BY 1"); 2
      // set cursor name for the positioned update statement
      pstmt1.setCursorName("c1");                                        3
      pstmt1.setString(1, "STAFF");
      ResultSet rs = pstmt1.executeQuery();                              4

      System.out.print("\n");
      while( rs.next() )                                                 5
      {   String tableName = rs.getString("tabname");
        System.out.println("Table = " + tableName);
      };
```

```
            rs.close();
            pstmt1.close();                                                         7
            }
            catch( Exception e )
            {   throw e;
            }
            finally
            {   // Rollback the transaction
              System.out.println("\nRollback the transaction...");
              con.rollback();
              System.out.println("Rollback done.");
            }
          }
          catch( Exception e )
          {   System.out.println(e);
          }
        }
      }
```

COBOL 예: DYNAMIC.SQB

```
       Identification Division.
       Program-ID. "dynamic".

       Data Division.
       Working-Storage Section.

           copy "sqlenv.cbl".
           copy "sql.cbl".
           copy "sqlca.cbl".

           EXEC SQL BEGIN DECLARE SECTION END-EXEC.
       01 table-name      pic x(20).
       01 st              pic x(80).     1
       01 parm-var        pic x(18).
       01 userid            pic x(8).
       01 passwd.
         49 passwd-length   pic s9(4) comp-5 value 0.
         49 passwd-name     pic x(18).
           EXEC SQL END DECLARE SECTION END-EXEC.

       77 errloc          pic x(80).

       Procedure Division.
       Main Section.
           display "Sample COBOL program: DYNAMIC".

           display "Enter your user id (default none): "
                with no advancing.
           accept userid.

           if userid = spaces
             EXEC SQL CONNECT TO sample END-EXEC
           else
             display "Enter your password : " with no advancing
             accept passwd-name.

      * Passwords in a CONNECT statement must be entered in a VARCHAR format
      * with the length of the input string.
           inspect passwd-name tallying passwd-length for characters
              before initial " ".

           EXEC SQL CONNECT TO sample USER :userid USING :passwd
               END-EXEC.
           move "CONNECT TO" to errloc.
           call "checkerr" using SQLCA errloc.

           move "SELECT TABNAME FROM SYSCAT.TABLES
      -       " ORDER BY 1
      -       " WHERE TABNAME <> ?" to st.
           EXEC SQL PREPARE s1 FROM :st END-EXEC.   2
           move "PREPARE" to errloc.
           call "checkerr" using SQLCA errloc.

           EXEC SQL DECLARE c1 CURSOR FOR s1 END-EXEC.   3

           move "STAFF" to parm-var.
           EXEC SQL OPEN c1 USING :parm-var END-EXEC.    4
           move "OPEN" to errloc.
           call "checkerr" using SQLCA errloc.
```

```
    perform Fetch-Loop thru End-Fetch-Loop
       until SQLCODE not equal 0.

    EXEC SQL CLOSE c1 END-EXEC.     6
    move "CLOSE" to errloc.
    call "checkerr" using SQLCA errloc.

    EXEC SQL COMMIT END-EXEC.
    move "COMMIT" to errloc.
    call "checkerr" using SQLCA errloc.

    EXEC SQL CONNECT RESET END-EXEC.
    move "CONNECT RESET" to errloc.
    call "checkerr" using SQLCA errloc.

End-Main.
    go to End-Prog.

Fetch-Loop Section.
    EXEC SQL FETCH c1 INTO :table-name END-EXEC.    5
    if SQLCODE not equal 0
       go to End-Fetch-Loop.
    display "TABLE = ", table-name.
End-Fetch-Loop. exit.

End-Prog.
    stop run.
```

REXX 예: DYNAMIC.CMD

```
/* REXX DYNAMIC.CMD */

parse version rexxType .
parse source platform .

if platform == 'AIX/6000' & rexxType == 'REXXSAA' then
do
  rcy = SysAddFuncPkg("db2rexx")
end
else
do
  if RxFuncQuery('SQLDBS') <> 0 then
    rcy = RxFuncAdd( 'SQLDBS',  'db2ar', 'SQLDBS'  )

  if RxFuncQuery('SQLEXEC') <> 0 then
    rcy = RxFuncAdd( 'SQLEXEC', 'db2ar', 'SQLEXEC' )
end

/* pull in command line arguments */
parse arg userid passwd .

/* check to see if the proper number of arguments have been passed in */
   PARSE ARG dbname userid password .
   if ((dbname = "" ) | ,
       (userid <> "" & password = "") ,
      ) then do
      SAY "USAGE: dynamic.cmd <dbname> [<userid> <password>]"

       exit -1
   end

   /* connect to database */
   SAY
   SAY 'Connect to' dbname
   IF password= "" THEN
      CALL SQLEXEC 'CONNECT TO' dbname
   ELSE
      CALL SQLEXEC 'CONNECT TO' dbname 'USER' userid 'USING' password

   CALL CHECKERR 'Connect to '
   SAY "Connected"

say 'Sample REXX program: DYNAMIC'

st = "SELECT tabname FROM syscat.tables WHERE tabname <> ? ORDER BY 1"
call SQLEXEC 'PREPARE s1 FROM :st'   2
```

```
call CHECKERR 'PREPARE'

call SQLEXEC 'DECLARE c1 CURSOR FOR s1'   3
call CHECKERR 'DECLARE'

parm_var = "STAFF"
call SQLEXEC 'OPEN c1 USING :parm_var'   4

do while ( SQLCA.SQLCODE = 0 )
  call SQLEXEC 'FETCH c1 INTO :table_name'   5
  if (SQLCA.SQLCODE = 0) then
    say 'Table = ' table_name
end

call SQLEXEC 'CLOSE c1'   6
call CHECKERR 'CLOSE'

call SQLEXEC 'CONNECT RESET'
call CHECKERR 'CONNECT RESET'


CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else do
    say '--- error report ---'
    say 'ERROR occurred :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /******************************\
    * GET ERROR MESSAGE API called *
    \******************************/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else do
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
return 0
```

SQLDA 선언

SQLDA에는 SQLVAR 항목 발생의 변수 숫자가 포함되어 있으며, SQLVAR 항목 각각에는 그림2에 표시된 바와 같이 한 행의 데이터에 하나의 컬럼을 설명하는 필드 세트가 들어 있습니다. 기본 SQLVAR과 2차 SQLVAR, 두 가지 유형의 SQLVAR 항목이 있습니다. 두 유형에 대한 설명은 *SQL 참조서*의 내용을 참조하십시오.

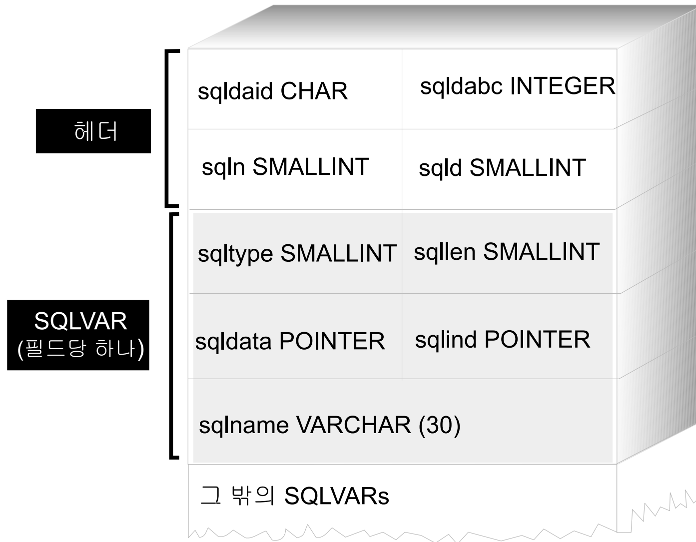


그림 2. SQL 설명자 영역(SQLDA)

필요한 SQLVAR 항목의 수는 결과 테이블의 컬럼 수에 따라 달라지므로, 응용프로그램은 필요할 때 적절한 수의 SQLVAR 요소를 할당해야 합니다. 아래에 사용가능한 두 가지 메소드에 대해 설명되어 있습니다. 언급된 SQLDA 필드에 대해서는 *SQL 참조서*의 내용을 참조하십시오.

- 필요한 최대 SQLDA(즉, SQLVAR 항목 수가 가장 많은 SQLDA)를 제공하십시오. 하나의 결과 테이블에서 리턴될 수 있는 최대 컬럼 수는 255입니다. 리턴 중인 컬럼이 LOB 유형이거나 구별 유형인 경우, SQLN의 값은 두 배로 되고, 정보를 보유하는 데 필요한 SQLVAR 수는 510으로 2배가 됩니다. 그러나, 대부분의 SELECT문이 255개의 컬럼도 검색하지 않으므로 할당된 공간의 대부분이 사용되지 않습니다.
- 더 작은 SQLDA에 더 적은 수의 SQLVAR 항목을 제공합니다. 이런 경우, SQLDA에서 허용된 SQLVAR 항목보다 많은 컬럼이 결과 테이블에 있는 경우, 설명은 리턴되

지 않습니다. 대신, 데이터베이스 관리 프로그램은 SELECT문에서 검출된 선택 목록 항목 수를 리턴합니다. 응용프로그램이 필요한 수의 SQLVAR 항목으로 SQLDA를 할당한 후, DESCRIBE문을 사용하여 컬럼 설명을 확보합니다. 이 메소드에 대한 자세한 설명은 『최소 SQLDA 구조를 사용하는 명령문 준비』에 있습니다.

위 메소드의 경우, 할당해야 하는 초기 SQLVAR 항목 수에 관한 문제가 발생할 수 있습니다. 각 SQLVAR 요소는 44바이트의 저장영역을 사용합니다.(이 저장영역은 SQLDATA 및 SQLIND 필드에 대해 할당된 것을 계산되지 않습니다.) 메모리가 충분할 경우, 최대 크기의 SQLDA를 제공하는 첫번째 방법을 구현하기가 더 쉽습니다.

더 작은 SQLDA를 할당하는 두 번째 메소드는 동적 메모리 할당을 지원하는 C 및 C++와 같은 프로그래밍 언어에만 적용될 수 있습니다. 동적 메모리 할당을 지원하지 않는 COBOL 및 FORTRAN과 같은 언어의 경우, 첫번째 메소드를 사용해야 합니다.

최소 SQLDA 구조를 사용하는 명령문 준비

응용프로그램이 SQLVAR 항목이 포함되어 있지 않은 minsqlda라는 SQLDA 구조를 선언한다고 합시다. SQLDA의 SQLN 필드는 할당된 SQLVAR 항목 수에 대해 설명합니다. 이런 경우, SQLN은 0으로 설정되어야 합니다. 그 다음, dstring 문자열로부터 명령문을 준비하고 그 설명을 minsqlda에 입력하려면, 다음 SQL문을 실행하십시오. (C구문이라고 하고, minsqlda가 SQLDA 구조에 대한 포인터로서 선언되어 있다고 가정합니다.)

```
EXEC SQL
  PREPARE STMT INTO :*minsqlda FROM :dstring;
```

dstring에 포함된 명령문이 각 행에서 20개 컬럼을 리턴하는 SELECT문이었다고 합시다. PREPARE문(또는 DESCRIBE문) 다음에 SQLDA의 SQLD 필드에는 준비된 SELECT문에 대한 결과 테이블의 컬럼 수가 들어 있습니다.

SQLDA에 있는 SQLVAR은 다음과 같은 경우 설정됩니다.

- SQLN >= SQLD 및 컬럼은 LOB이거나 구별 유형이 아닙니다.

  첫번째 SQLD SQLVAR 항목이 설정되었고 SQLDOUBLED는 공백으로 설정됩니다.

- SQLN >= 2*SQLD이고, 적어도 한 컬럼이 LOB이거나 구별 유형입니다.

  2* SQLD SQLVAR 항목이 설정되고, SQLDOUBLED가 2로 설정됩니다.

- SQLD <= SQLN < 2*SQLD이고, 적어도 한 컬럼이 구별 유형이지만 LOB 컬럼이 없습니다.

  첫번째 SQLD SQLVAR 항목이 설정되었고 SQLDOUBLED는 공백으로 설정됩니다. SQLWARN 바인드 옵션이 YES인 경우 경고 SQLCODE +237 (SQLSTATE 01594)이 발행됩니다.

다음과 같은 경우 SQLDA의 SQLVAR이 설정되지 않습니다(추가 공간 및 또다른 DESCRIBE의 할당이 필요).

- SQLN < SQLD이고, 컬럼은 LOB이거나 구별 유형이 아닙니다.

  SQLVAR 항목이 설정되지 않고 SQLDOUBLED가 공백으로 설정됩니다. SQLWARN 바인드 옵션이 YES인 경우, 경고 SQLCODE +236(SQLSTATE 01005)이 실행됩니다.

  성공적인 DESCRIBE를 위해 SQLD SQLVAR을 할당하십시오.

- SQLN < SQLD이고, 적어도 한 컬럼이 구별 유형이지만 LOB 컬럼은 없습니다.

  SQLVAR 항목이 설정되지 않고 SQLDOUBLED가 공백으로 설정됩니다. SQLWARN 바인드 옵션이 YES이고, 경고 SQLCODE +239(SQLSTATE 01005)가 실행됩니다.

  구별 유형의 이름을 포함하여, 성공적인 DESCRIBE를 위해 2*SQLD SQLVAR을 할당합니다.

- SQLN < 2*SQLD이고, 적어도 하나의 컬럼이 LOB입니다.

  SQLVAR 항목이 설정되지 않고 SQLDOUBLED가 공백으로 설정됩니다. (SQLWARN 바인드 옵션 설정에 관계없이) 경고 SQLCODE +238(SQLSTATE 01005)이 실행됩니다.

  성공적인 DESCRIBE를 위해 2*SQLD SQLVAR을 할당합니다.

BIND 명령의 SQLWARN 옵션을 사용하여 DESCRIBE(또는 PREPARE...INTO)가 다음 경고를 리턴할 것인지 통제합니다.

- SQLCODE +236 (SQLSTATE 01005)
- SQLCODE +237 (SQLSTATE 01594)
- SQLCODE +239 (SQLSTATE 01005).

사용자의 응용프로그램 코드가 항상 이러한 SQLCODE를 리턴할 수 있도록 하는 것이 좋습니다. 선택 목록에 LOB 컬럼이 있고 SQLDA에 충분치 못한 SQLVAR이 있을 때 항상 경고 SQLCODE +238(SQLSTATE 01005)이 리턴됩니다. 이는 응용프로그램이 결과 세트의 LOB 컬럼으로 인해 SQLVAR 수가 2배로 되어야 함을 알 수 있는 유일한 방법입니다.

충분한 SQLVAR 항목으로 SQLDA 할당

결과 테이블의 컬럼 수가 결정되면 두 번째의 전체 크기 SQLDA에 대한 저장영역을 할당할 수 있습니다. 예를 들어, 결과 테이블에 20개 컬럼이 있는 경우(이 중 LOB 컬럼은 없음), 두 번째 SQLDA 구조인 `fulsqlda`가 적어도 20개의 SQLVAR 요소가 할당되어야 합니다(결과 테이블에 LOB 또는 구별 유형이 포함되어 있는 경우는 40개). 이 예의 나머지에서는 LOB이나 구별 유형이 결과 테이블에 없다고 가정합니다.

SQLDA 구조용 저장영역 조건은 다음과 같습니다.

- SQLN 및 SQLD와 같은 필드가 포함된 16바이트의 고정 길이 헤더
- SQLVAR의 항목의 가변 길이 배열은 32- 비트 플랫폼에서 44 바이트 그리고 64- 비트 플랫폼에서 56 바이트입니다.

`fulsqlda`에 필요한 SQLVAR 항목의 수는 `minsqlda`의 SQLD 필드에 지정되어 있습니다. 이 값은 20입니다. 따라서, 이 예에서 사용된 `fulsqlda`에 필요한 저장영역 할당은 다음과 같습니다.

```
16 + (20 * sizeof(struct sqlvar))
```

주: 64- 비트 플랫폼에서, `sizeof(struct sqlvar)` 및 `sizeof(struct sqlvar2)`는 56을 리턴합니다. 32- 비트 플랫폼에서는 `sizeof(struct sqlvar)` 및 `sizeof(struct sqlvar2)`는 44를 리턴합니다.

이 값은 전체 896바이트를 제공하는 각 SQLVAR 항목 크기의 20배에 헤더 크기를 더한 값입니다.

SQLDASIZE 매크로를 사용하여 사용자 자신의 계산을 피하고, 버전별 종속성을 피할 수 있습니다.

SELECT문 설명

fulsqlda 및 응용프로그램에 대해 충분한 공간을 할당할 때 다음 단계를 따라야 합니다.

1. fulsqlda의 SQLN 필드에 20 값을 저장합니다.
2. 두 번째 SQLDA 구조인 fulsqlda를 사용하여 SELECT문에 대한 정보를 얻습니다. 두 가지 방법이 사용가능합니다.
   - minsqlda 대신 fulsqlda를 사용하는 다른 PREPARE문을 사용합니다.
   - fulsqlda를 지정하는 DESCRIBE문을 사용합니다.

명령문을 두 번 준비하는 비용이 삼가되므로 DESCRIBE문을 사용하는 것이 좋습니다. DESCRIBE문은 새로운 SQLDA 구조에 들어갈 준비 조작을 하는 동안 이전에 얻은 정보를 재사용합니다. 다음 명령문이 발행될 수 있습니다.

```
EXEC SQL DESCRIBE STMT INTO :fulsqlda
```

이 명령문이 실행된 후 각 SQLVAR 요소에는 결과 테이블의 한 컬럼에 대한 설명이 들어 있습니다.

행 보유를 위한 저장영역 취득

SQLDA 구조를 사용한 결과 테이블 행을 페치하기 전에 응용프로그램은 다음을 수행해야 합니다.

1. 각 SQLVAR 설명을 분석하여 해당 컬럼 값에 어느 정도의 공간이 필요한지 결정합니다.

   대형 오브젝트(LOB) 값의 경우 SELECT가 설명될 때 SQLVAR에 제공되는 데이터 유형은 SQL_TYP_*x*LOB입니다. 이 데이터 유형은 일반 LOB 호스트 변수에 일치합니다. 즉, 전체 LOB가 한 번에 메모리에 저장됩니다. 이는 (서너 개의 MB까지) 소형 LOB에 대해 작동되지만 대형 LOB(1GB라고 하면)에 대해 이 데이터 유형을 사용할 수 없습니다. 응용프로그램이 SQLVAR에서 컬럼 정의가 SQL_TYP_*x*LOB_LOCATOR 또는 SQL_TYPE_*x*LOB_FILE로 변경되어야 합니다.(SQLVAR의 SQLTYPE 필드를 변경하면 SQLLEN 필드도 변경해야 합니다.) SQLVAR에서의 컬럼 정의를 변경한 후 응용프로그램은 새로운 유형에 대해 올바른 크기의 저장영역을

할당할 수 있습니다. LOB에 대한 세부사항은 297 페이지의『제10장 오브젝트 관계형 기능 사용』의 내용을 참조하십시오.

2. 해당 컬럼의 값에 대한 저장영역을 할당합니다.
3. SQLDA 구조의 SQLDATA 필드에 할당된 저장영역의 주소를 저장합니다.

이 단계는 각 컬럼의 설명을 분석하고 각 SQLDATA 필드의 내용을 해당 컬럼의 값을 보유하기에 충분한 크기의 저장영역의 주소로 대체하여 이루어집니다. 길이 속성은 LOB 유형이 아닌 데이터 항목에 대한 각 SQLVAR의 SQLLEN 필드에서 결정됩니다. 유형이 BLOB, CLOB 또는 DBCLOB인 항목의 경우, 길이 속성은 2차 SQLVAR 항목의 SQLLONGLEN 필드로부터 결정됩니다.

또한, 지정된 컬럼이 널(NULL)을 허용할 경우, 응용프로그램은 SQLIND 필드 내용을 컬럼에 대한 표시기 변수 주소로 대체해야 합니다.

커서 처리

SQLDA 구조가 제대로 할당되면 SELECT문과 연관된 커서를 열 수 있고, FETCH 문의 USING DESCRIPTOR절을 지정하여 행을 페치할 수 있습니다.

완료가 되면 커서를 닫고 동적 할당된 메모리를 릴리스해야 합니다.

SQLDA 구조 할당

C로 SQLDA 구조를 작성하려면 호스트 언어로 INCLUDE SQLDA문을 포함시키거나, 구조 정의 확보를 위해 SQLDA 포함 파일을 포함시키십시오. 그런 후, SQLDA 크기가 고정되어 있지 않으므로 응용프로그램은 SQLDA 구조에 대한 포인터를 선언하고 저장영역을 할당해야 합니다. SQLDA 구조의 실제 크기는 SQLDA를 사용하여 전달되는 구별 데이터 항목의 수에 따라 달라집니다.(SQLDA 처리를 위해 응용프로그램을 코드화하는 방법의 예는 174 페이지의『예: ADHOC 프로그램』의 내용을 참조하십시오.)

C/C++ 프로그래밍 언어에서, 매크로는 SQLDA 할당을 가능하게 하기 위해 제공됩니다. HP-UX 플랫폼의 예외로 이 매크로는 다음의 형식을 갖습니다.

```
#define SQLDASIZE(n) (offsetof(struct sqlda, sqlvar) + (n) × sizeof(struct sqlvar))
```

HP-UX 플랫폼에서 매크로는 다음의 형식을 갖습니다.

```
#define SQLDASIZE(n) (sizeof(struct sqlda) + (n-1) × sizeof(struct sqlvar))
```

이 매크로의 효과는 SQLDA의 필요한 저장영역을 n개의 SQLVAR 요소를 이용하여 계산하는 것입니다.

COBOL을 사용하여 SQLDA 구조를 작성하기 위해 INCLUDE SQLDA문을 포함시키거나 COPY문을 사용할 수 있습니다. 최대 수의 SQLVAR과 SQLDA가 사용하는 저장영역 크기를 제어하려면 COPY문을 사용하십시오. 예를 들어, SQLVAR의 기본값을 1489에서 1로 변경하려면 다음과 같은 COPY문을 사용하십시오.

```
COPY "sqlda.cbl"
  replacing --1489--
  by --1--.
```

FORTRAN 언어는 자동 정의 데이터 구조 또는 동적 할당을 직접 지원하지 않습니다. FORTRAN의 경우 SQLDA 포함 파일이 제공되지 않습니다. FORTRAN으로 데이터 구조로서 SQLDA를 지원하는 것이 가능하지 않기 때문입니다. 사전처리 컴파일러는 FORTRAN 프로그램에서 INCLUDE SQLDA문을 무시합니다.

그러나, FORTRAN 프로그램에서 정적 SQLDA 구조와 유사한 구조를 작성할 수 있으며, SQLDA를 사용할 수 있는 곳에서 이 구조를 사용할 수 있습니다. `sqldact.f` 파일에는 FORTRAN으로 SQLDA 구조를 선언하는 데 도움이 되는 상수가 포함되어 있습니다.

필요한 SQLDA 요소에 포인터 값을 지정하기 위해 SQLGADDR에 대한 호출을 실행합니다.

다음 테이블은 하나의 SQLDA 요소를 가진 SQLDA 구조의 선언 및 사용을 표시합니다.

언어	예제 소스 코드

C/C++	`#include <sqlda.h>`
	`struct sqlda *outda = (struct sqlda *)malloc(SQLDASIZE(1));`
	
	`/* DECLARE LOCAL VARIABLES FOR HOLDING ACTUAL DATA */`
	`double sal;`
	`short salind;`
	
	`/* INITIALIZE ONE ELEMENT OF SQLDA */`
	`memcpy( outda->sqldaid,"SQLDA   ",sizeof(outda->sqldaid));`
	`outda->sqln = outda->sqld = 1;`
	`outda->sqlvar[0].sqltype = SQL_TYP_NFLOAT;`
	`outda->sqlvar[0].sqllen  = sizeof( double );.`
	`outda->sqlvar[0].sqldata = (unsigned char *)&sal;`
	`outda->sqlvar[0].sqlind  = (short *)&salind;`

COBOL	`WORKING-STORAGE SECTION.`
	`77 SALARY          PIC S99999V99 COMP-3.`
	`77 SAL-IND         PIC S9(4)     COMP-5.`
	
	`   EXEC SQL INCLUDE SQLDA END-EXEC`
	
	`* Or code a useful way to save unused SQLVAR entries.`
	`* COPY "sqlda.cbl" REPLACING --1489-- BY --1--.`
	
	`     01 decimal-sqllen pic s9(4) comp-5.`
	`     01 decimal-parts redefines decimal-sqllen.`
	`        05 precision pic x.`
	`        05 scale pic x.`
	
	`* Initialize one element of output SQLDA`
	`     MOVE 1 TO SQLN`
	`     MOVE 1 TO SQLD`
	`     MOVE SQL-TYP-NDECIMAL TO SQLTYPE(1)`
	
	`* Length = 7 digits precision and 2 digits scale`
	`     MOVE x"07" TO PRECISION.`
	`     MOVE x"02" TO SCALE.`
	`     MOVE DECIMAL-SQLLEN TO O-SQLLEN(1).`
	`     SET SQLDATA(1) TO ADDRESS OF SALARY`
	`     SET SQLIND(1)  TO ADDRESS OF SAL-IND`

언어	예제 소스 코드
FORTRAN	(see code below)

```
      include 'sqldact.f'

      integer*2  sqlvar1
      parameter ( sqlvar1 = sqlda_header_sz + 0*sqlvar_struct_sz )

C     Declare an Output SQLDA -- 1 Variable
      character    out_sqlda(sqlda_header_sz + 1*sqlvar_struct_sz)

      character*8  out_sqldaid     ! Header
      integer*4    out_sqldabc
      integer*2    out_sqln
      integer*2    out_sqld

      integer*2    out_sqltype1    ! First Variable
      integer*2    out_sqllen1
      integer*4    out_sqldata1
      integer*4    out_sqlind1
      integer*2    out_sqlnamel1
      character*30 out_sqlnamec1

      equivalence( out_sqlda(sqlda_sqldaid_ofs), out_sqldaid )
      equivalence( out_sqlda(sqlda_sqldabc_ofs), out_sqldabc )
      equivalence( out_sqlda(sqlda_sqln_ofs), out_sqln       )
      equivalence( out_sqlda(sqlda_sqld_ofs), out_sqld       )
      equivalence( out_sqlda(sqlvar1+sqlvar_type_ofs), out_sqltype1 )
      equivalence( out_sqlda(sqlvar1+sqlvar_len_ofs), out_sqllen1   )
      equivalence( out_sqlda(sqlvar1+sqlvar_data_ofs), out_sqldata1 )
      equivalence( out_sqlda(sqlvar1+sqlvar_ind_ofs), out_sqlind1   )
      equivalence( out_sqlda(sqlvar1+sqlvar_name_length_ofs),
     +             out_sqlnamel1                                   )
      equivalence( out_sqlda(sqlvar1+sqlvar_name_data_ofs),
     +             out_sqlnamec1                                   )

C     Declare Local Variables for Holding Returned Data.
         real*8       salary
      integer*2    sal_ind

C     Initialize the Output SQLDA (Header)
      out_sqldaid  = 'OUT_SQLDA'
      out_sqldabc  = sqlda_header_sz + 1*sqlvar_struct_sz
      out_sqln     = 1
      out_sqld     = 1
C     Initialize VAR1
      out_sqltype1 = SQL_TYP_NFLOAT
      out_sqllen1  = 8
      rc = sqlgaddr( %ref(salary), %ref(out_sqldata1) )
      rc = sqlgaddr( %ref(sal_ind), %ref(out_sqlind1) )
```

동적 메모리 할당을 지원하지 않는 언어로, 원하는 수의 SQLVAR 요소를 가진 SQLDA는 호스트 언어로 명시적으로 선언되어야 합니다. 응용프로그램의 필요에 의해 충분한 SQLVAR 요소를 선언하도록 하십시오.

SQLDA 구조를 사용한 데이터 전달

SQLDA를 사용한 데이터 전달하면 호스트 변수 목록을 사용할 때보다 융통성이 증가됩니다. 예를 들어, C 언어로 된 DECIMAL 데이터와 같은 본래의 호스트 언어가 없는 데이터를 전송하기 위해 SQLDA를 사용할 수 있습니다. ADHOC라는 샘플 프로그램은 이 기술을 사용하는 예입니다(174 페이지의 『예: ADHOC 프로그램』 참조). 수치 값 및 기호 이름이 어떻게 연관되어 있는지 표시하는 편리한 상호참조 목록은 표8의 내용을 참조하십시오.

표 8. DB2 V2 SQLDA SQL 유형. 숫자 값과 이에 상응하는 기호 이름

SQL 컬럼 유형	SQLTYPE 숫자 값	SQLTYPE 기호 이름[1]
DATE	384/385	SQL_TYP_DATE / SQL_TYP_NDATE
TIME	388/389	SQL_TYP_TIME / SQL_TYP_NTIME
TIMESTAMP	392/393	SQL_TYP_STAMP / SQL_TYP_NSTAMP
n/a[2]	400/401	SQL_TYP_CGSTR / SQL_TYP_NCGSTR
BLOB	404/405	SQL_TYP_BLOB / SQL_TYP_NBLOB
CLOB	408/409	SQL_TYP_CLOB / SQL_TYP_NCLOB
DBCLOB	412/413	SQL_TYP_DBCLOB / SQL_TYP_NDBCLOB
VARCHAR	448/449	SQL_TYP_VARCHAR / SQL_TYP_NVARCHAR
CHAR	452/453	SQL_TYP_CHAR / SQL_TYP_NCHAR
LONG VARCHAR	456/457	SQL_TYP_LONG / SQL_TYP_NLONG
n/a[3]	460/461	SQL_TYP_CSTR / SQL_TYP_NCSTR
VARGRAPHIC	464/465	SQL_TYP_VARGRAPH / SQL_TYP_NVARGRAPH
GRAPHIC	468/469	SQL_TYP_GRAPHIC / SQL_TYP_NGRAPHIC
LONG VARGRAPHIC	472/473	SQL_TYP_LONGRAPH / SQL_TYP_NLONGRAPH
FLOAT	480/481	SQL_TYP_FLOAT / SQL_TYP_NFLOAT
REAL[4]	480/481	SQL_TYP_FLOAT / SQL_TYP_NFLOAT
DECIMAL[5]	484/485	SQL_TYP_DECIMAL / SQL_TYP_DECIMAL
INTEGER	496/497	SQL_TYP_INTEGER / SQL_TYP_NINTEGER
SMALLINT	500/501	SQL_TYP_SMALL / SQL_TYP_NSMALL
n/a	804/805	SQL_TYP_BLOB_FILE / SQL_TYPE_NBLOB_FILE
n/a	808/809	SQL_TYP_CLOB_FILE / SQL_TYPE_NCLOB_FILE
n/a	812/813	SQL_TYP_DBCLOB_FILE / SQL_TYPE_NDBCLOB_FILE
n/a	960/961	SQL_TYP_BLOB_LOCATOR / SQL_TYP_NBLOB_LOCATOR
n/a	964/965	SQL_TYP_CLOB_LOCATOR / SQL_TYP_NCLOB_LOCATOR

표 8. *DB2 V2 SQLDA SQL 유형 (계속). 숫자 값과 이에 상응하는 기호 이름*

SQL 컬럼 유형	SQLTYPE 숫자 값	SQLTYPE 기호 이름[1]
n/a	968/969	SQL_TYP_DBCLOB_LOCATOR / SQL_TYP_NDBCLOB_LOCATOR

주: 정의된 유형은 sqllib 디렉토리의 include 서브디렉토리에 있는 sql.h 포함 파일에 있습니다. (예, C 프로그래밍 언어에서 sqllib/include/sql.h)

1. COBOL 프로그래밍 언어에서는 SQLTYPE 이름은 밑줄(_)을 사용하지 않고 대신 하이픈(-)을 사용합니다.
2. 널(null) 종료 그래픽 문자열입니다.
3. 널(null) 종료 문자열입니다.
4. SQLDA에서 DOUBLE과 REAL은 길이 값(4 또는 8)에서 차이가 납니다.
5. 정밀도는 첫번째 바이트입니다. 스케일은 두 번째 바이트입니다.

대화식 SQL문 처리

동적 SQL을 사용하는 응용프로그램을 작성하여 임의 SQL문을 처리할 수 있습니다. 예를 들어, 응용프로그램이 사용자로부터 SQL문을 허용할 경우, 응용프로그램은 명령문에 대한 이전 지식 없이 명령문을 실행할 수 있어야 합니다.

SQLDA 구조에서 PREPARE 및 DESCRIBE문을 사용하여 응용프로그램은 실행 중인 SQL문 유형을 결정하고 그에 따라 조작을 수행할 수 있습니다.

대화식 SQL문을 처리하는 프로그램의 예를 보려면 174 페이지의『예: ADHOC 프로그램』의 내용을 참조하십시오.

명령문 유형 결정

SQL문이 준비될 때 명령문 유형과 관련된 정보는 SQLDA 구조를 검토하여 결정됩니다. 이 정보는 INTO절에서 명령문이 준비될 때 또는 이전에 준비된 명령문에 대해 DESCRIBE문을 발행하여 SQLDA 구조에 둘 수 있습니다.

어떤 경우이든 데이터베이스 관리 프로그램은 값을 구조의 SQLD 필드에 값을 둡니다. 이 값은 SQL문에 의해 생성된 결과 테이블에 있는 컬럼 수를 나타냅니다. SQLD 필드에 제로(0) 값이 포함되어 있을 경우 명령문은 SELECT문이 *아닙니다*. 명령문이 이미 준비되었으므로 EXECUTE문을 사용하여 즉시 실행될 수 있습니다.

명령문에 매개변수 표시문자가 포함되어 있는 경우 USING절을 *SQL 참조서*에서 와 같이 지정해야 합니다. USING절은 호스트 변수 목록 또는 SQLDA 구조를 지정할 수 있습니다.

SQLD 필드가 제로보다 클 경우 명령문은 SELECT문이고, 다음 절에서 설명된 대로 처리해야 합니다.

Varying-List SELECT문

varying-list SELECT문은 리턴되는 컬럼 유형 및 컬럼 수가 사전 처리 컴파일될 때 알려지지 않는 명령문입니다. 이런 경우 응용프로그램은 결과 테이블의 행을 보유하도록 선언해야 할 호스트 변수를 미리 알지 못합니다.

varying-list SELECT문을 처리하기 위해 응용프로그램은 다음을 수행할 수 있습니다.

1. **SQLDA 선언.** varying-list SELECT문을 처리하기 위해 SQLDA 구조를 사용해야 합니다.
2. **INTO절을 사용한 명령문 PREPARE.** 그런 후 응용프로그램은 선언된 SQLDA 구조에 충분한 SQLVAR 요소가 있는지 결정합니다. 그렇지 않은 경우, 응용프로그램은 필요한 수의 SQLVAR 요소를 가진 다른 SQLDA 구조를 할당하고 새로운 SQLDA를 사용하여 추가 DESCRIBE문을 실행합니다.
3. **SQLVAR 요소 할당.** 각 SQLVAR에 필요한 호스트 변수 및 표시기에 대한 저장영역을 할당합니다. 이 단계에는 데이터 및 표시기 변수에 대해 할당된 주소를 각 SQLVAR 요소에 둡니다.
4. **SELECT문 처리.** 커서는 준비된 명령문과 연관 및 열려 있고, 행들은 적절히 할당된 SQLDA 구조를 사용하여 페치됩니다.

이 단계에 대해서는 다음 절에서 상세히 설명합니다.

- 162 페이지의『SQLDA 선언』
- 163 페이지의『최소 SQLDA 구조를 사용하는 명령문 준비』
- 165 페이지의『충분한 SQLVAR 항목으로 SQLDA 할당』
- 166 페이지의『SELECT문 설명』
- 166 페이지의『행 보유를 위한 저장영역 취득』
- 167 페이지의『커서 처리』.

일반 사용자로부터 SQL 요청 저장

응용프로그램을 이용하여 임의의 SQL문을 저장할 수 있는 경우, 데이터 유형이 VARCHAR, LONG VARCHAR, CLOB, VARGRAPHIC, LONG VARGRAPHIC 또는 DBCLOB인 컬럼을 가진 테이블에 저장할 수 있습니다. VARGRAPHIC, LONG VARGRAPHIC 및 DBCLOB 데이터 유형은 DBCS(2 바이트 문자 지원) 및 EUC(확장 UNIX 코드) 환경에서만 사용가능합니다.

준비된 버전이 아닌 소스 SQL문을 저장해야 합니다. 이는 테이블에 저장된 버전을 실행하기 전에 각 명령문을 검색한 후 준비해야 한다는 것을 의미합니다. 근본적으로 사용자 응용프로그램은 문자열로부터 SQL문을 준비하고 이 명령문을 동적으로 실행합니다.

예: ADHOC 프로그램

이 샘플 프로그램은 SQLDA를 사용하여 대화식 SQL문을 처리하는 방법을 보여줍니다.

주: 예 adhoc.sqc는 C 언어에만 있습니다.

ADHOC 프로그램 작업 방법

1. **SQLDA 구조 정의** INCLUDE SQLDA문은 SQLDA 구조를 정의 및 선언하며, 이를 사용하여 데이터베이스 관리 프로그램에서 프로그램으로, 또는 프로그램에서 데이터베이스 관리 프로그램을 데이터를 전달합니다.
2. **SQLCA 구조 정의** INCLUDE SQLCA문은 SQLCA 구조를 정의하고, 구조 내의 요소로서 SQLCODE를 정의합니다. SQLCA 구조의 SQLCODE 필드는 SQL문을 실행한 후 데이터베이스 관리 프로그램에 의한 진단 정보로 갱신됩니다.
3. **호스트 변수 정의.** BEGIN DECLARE SECTION 및 END DECLARE SECTION문은 호스트 변수 선언을 분리합니다. 호스트 변수는 SQL문에서 참조될 때 콜론(:)으로 시작됩니다.
4. **데이터베이스에 연결.** 프로그램은 사용자가 지정한 데이터베이스에 연결하여 이에 대한 공유 액세스를 요청합니다.(START DATABASE MANAGER

API 호출이나 db2start 명령이 발행되었다고 전제합니다.) 공유 모드로 동일한 데이터베이스에 연결하려고 하는 기타 프로그램에게도 액세스가 권한 부여됩니다.

5. **점검 완료.** SQLCA 구조에서 CONNECT TO문이 성공적으로 완료되었는지 점검됩니다. SQLCODE 값 0은 연결이 성공했음을 나타냅니다.
6. **대화식 프롬프트.** SQL문은 프롬프트를 통해 입력되며, process_statement 함수로 전송됩니다.
7. **트랜잭션 종료 - COMMIT.** 작업 단위(UOW)는 사용자가 선택할 경우 COMMIT으로 종료됩니다. 최종 COMMIT 이후 입력된 SQL문이 요청한 모든 변경사항은 데이터베이스에 저장됩니다.
8. **트랜잭션 종료 - ROLLBACK.** 작업 단위(UOW)는 사용자가 선택할 경우 ROLLBACK으로 종료됩니다. 최종 COMMIT 또는 프로그램 시작 이후 입력된 SQL문이 요청한 모든 변경사항은 실행 취소됩니다.
9. **데이터베이스로부터 연결해제.** 프로그램은 CONNECT RESET문을 실행하여 데이터베이스로부터의 연결을 해제합니다. 리턴될 때 SQLCA가 성공적으로 완료되었는지 점검됩니다.
10. **SQL문 텍스트를 호스트 변수로 복사.** 명령문 텍스트는 호스트 변수 st가 지정한 데이터 영역으로 복사됩니다.
11. **SQLDA 처리 준비.** init_da 프로시듀어를 통해 초기 SQLDA 구조가 선언되고 메모리가 할당되어, SQL문이 생성할 수 있는 출력 유형을 결정합니다. 이 PREPARE문으로부터 리턴된 SQLDA는 SQL문으로부터 리턴될 컬럼 수를 보고합니다.
12. **SQLDA 보고서 출력 컬럼이 존재합니다.** SQL문은 SELECT문입니다. SQLDA는 준비된 SQL문이 상주할 메모리 공간을 할당하기 위해 init_da 프로시듀어를 통해 초기화됩니다.
13. **SQLDA는 출력 컬럼을 보고하지 않습니다.** 리턴될 컬럼이 없습니다. SQL문은 EXECUTE문을 사용하여 동적으로 실행됩니다.
14. **SQLDA용 메모리 공간준비** SQLDA에서 컬럼 구조를 반영하도록 메모리가 할당됩니다. 필요한 양의 메모리는 SQLDA에 있는 컬럼 구조의 SQLLEN 및 SQLTYPE에 의해 선택됩니다.

15. **커서 선언 및 열기.** DECLARE문은 pcurs 커서를 sqlStatement에 있는 동적으로 준비된 SQL문을 연결하고 커서가 열려 있습니다.
16. **행 검색.** FETCH문은 다음 행에 커서를 두고 행 내용을 SQLDA로 이동시킵니다.
17. **컬럼 제목 표시.** 페치된 첫번째 행은 컬럼 제목 정보입니다.
18. **행 정보 표시.** 연속된 각 FETCH로부터 수집된 정보 행이 표시됩니다.
19. **커서 닫기.** CLOSE문은 커서를 닫고 이와 연결된 자원을 릴리스합니다.

EMB_SQL_CHECK 매크로/함수는 이프로그램의 외부에 있는 오류 점검 유틸리티입니다. DB2 API를 호출하는 C 프로그램에 대해서는 utilapi.c에서 sqlInfoPrint 함수는 utilapi.h에서 API_SQL_CHECK로서 참조됩니다. C embedded SQL 프로그램에 대해서는 utilemb.sqc에서 sqlInfoPrint 함수는 utilemb.h에서 EMB_SQL_CHECK로서 참조됩니다. 이 오류 체크 유틸리티에 대한 소스 코드에 대해서는 137 페이지의『예제 프로그램에서 GET ERROR MESSAGE 사용』의 내용을 참조하십시오.

이 예에서는 utilemb.sqc 파일의 유틸리티로서 제공되는 추가 프로시듀어를 사용합니다. 여기에는 다음이 포함됩니다.

init_da

준비된 SQL문에 대한 메모리를 할당합니다. SQLDASIZE라는 내부 서술 함수는 적절한 메모리량을 계산하는 데 사용됩니다.

alloc_host_vars

SQLDA 포인터로부터 데이터에 대한 메모리를 할당합니다.

free_da

SQLDA 데이터 구조를 사용하는 데 할당된 메모리를 비웁니다.

print_var

SQLDA SQLVAR 변수를 인쇄합니다. 우선 이 프로시듀어는 데이터 유형을 결정하고, 데이터를 인쇄하는 데 필요한 해당 서브루틴을 호출합니다.

display_da

전달된 포인터의 출력을 표시합니다. print_var 프로시듀어에서 검토된 대로, 출력 데이터의 구조에 관한 모든 정보를 이 포인터로부터 사용할 수 있습니다.

C 예: ADHOC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlcodes.h>
#include <sqlda.h>  1
#include "utilemb.h"

#ifdef DB268K
  /* Need to include ASLM for 68K applications */
  #include <LibraryManager.h>
#endif

EXEC SQL INCLUDE SQLCA ;  2

#define SQLSTATE sqlca.sqlstate

int process_statement( char * ) ;

int main( int argc, char *argv[] ) {

   int rc ;
   char sqlInput[256] ;
   char st[1024] ;

   EXEC SQL BEGIN DECLARE SECTION ;  3
      char userid[9] ;
      char passwd[19] ;
   EXEC SQL END DECLARE SECTION ;

#ifdef DB268K
  /*
   Before making any API calls for 68K environment,
   need to initial the Library Manager
   */
   InitLibraryManager(0,kCurrentZone,kNormalMemory) ;
   atexit(CleanupLibraryManager) ;
#endif

   printf( "Sample C program : ADHOC interactive SQL\n" ) ;

   /* Initialize the connection to a database. */
   if ( argc == 1 ) {
      EXEC SQL CONNECT TO sample ;
      EMB_SQL_CHECK( "CONNECT TO SAMPLE" ) ;
   }
   else if ( argc == 3 ) {
      strcpy( userid, argv[1] ) ;
      strcpy( passwd, argv[2] ) ;
      EXEC SQL CONNECT TO sample USER :userid USING :passwd ;  4
      EMB_SQL_CHECK( "CONNECT TO SAMPLE" ) ;  5
   }
   else {
      printf( "\nUSAGE: adhoc [userid passwd]\n\n" ) ;
      return( 1 ) ;
   } /* endif */

   printf( "Connected to database SAMPLE\n" ) ;

   /* Enter the continuous command line loop. */
   *sqlInput = '\0' ;
   while ( ( *sqlInput != 'q' ) && ( *sqlInput != 'Q' ) ) {  6

      printf( "Enter an SQL statement or 'quit' to Quit :\n" ) ;
```

```
      gets( sqlInput ) ;

      if ( ( *sqlInput == 'q' ) || ( *sqlInput == 'Q' ) ) break ;

      if ( *sqlInput == '\0' ) { /* Don't process the statement */
         printf( "No characters entered.\n" ) ;
         continue ;
      }

      strcpy( st, sqlInput ) ;
      while ( sqlInput[strlen( sqlInput ) - 1] == '\\' ) {
         st[strlen( st ) - 1] = '\0' ;
         gets( sqlInput ) ;
         strcat( st, sqlInput ) ;
      }

      /* Process the statement. */
      rc = process_statement( st ) ;

   }

   printf( "Enter 'c' to COMMIT or Any Other key to ROLLBACK the transaction :\n" ) ;
   gets( sqlInput ) ;
   if ( ( *sqlInput == 'c' ) || ( *sqlInput == 'C' ) ) {
      printf( "COMMITING the transactions.\n" ) ;
      EXEC SQL COMMIT ;  7
      EMB_SQL_CHECK( "COMMIT" ) ;
   }
   else { /* assume that the transaction is to be rolled back */
      printf( "ROLLING BACK the transactions.\n" ) ;
      EXEC SQL ROLLBACK ; 8
      EMB_SQL_CHECK( "ROLLBACK" ) ;
   }

   EXEC SQL CONNECT RESET ; 9
   EMB_SQL_CHECK( "CONNECT RESET" ) ;

   return( 0 ) ;

}

/******************************************************************
 * FUNCTION : process_statement
 * This function processes the inputted statement and then prepares the
 *  procedural SQL implementation to take place.
******************************************************************/
int process_statement ( char * sqlInput ) {

   int counter = 0 ;
   struct sqlda * sqldaPointer ;
   short sqlda_d ;

   EXEC SQL BEGIN DECLARE SECTION ; 3
      char st[1024] ;
   EXEC SQL END DECLARE SECTION ;

   strcpy( st, sqlInput ) ; 10
   /* allocate an initial SQLDA temp pointer to obtain information
      about the inputted "st" */

   init_da( &sqldaPointer, 1 ) ; 11

   EXEC SQL PREPARE statement1 from :st ;
   /* EMB_SQL_CHECK( "PREPARE" ) ; */

   EXEC SQL DESCRIBE statement1 INTO :*sqldaPointer ;
```

```
/* Expecting a return code of 0 or SQL_RC_W236,
   SQL_RC_W237, SQL_RC_W238, SQL_RC_W239 for cases
   where this statement is a SELECT statment. */
if ( SQLCODE != 0           &&
     SQLCODE != SQL_RC_W236 &&
     SQLCODE != SQL_RC_W237 &&
     SQLCODE != SQL_RC_W238 &&
     SQLCODE != SQL_RC_W239
   ) {
   /* An unexpected warning/error has occurred. Check the SQLCA. */
   EMB_SQL_CHECK( "DESCRIBE" ) ;
} /* end if */

sqlda_d = sqldaPointer->sqld ;
free( sqldaPointer ) ;

if ( sqlda_d > 0 ) {  12

   /* this is a SELECT statement, a number of columns
      are present in the SQLDA */

   if ( SQLCODE == SQL_RC_W236 || SQLCODE == 0)
      /* this out only needs a SINGLE SQLDA */
      init_da( &sqldaPointer, sqlda_d ) ;

   if ( SQLCODE == SQL_RC_W237 ||
        SQLCODE == SQL_RC_W238 ||
        SQLCODE == SQL_RC_W239 )
      /* this output contains columns that need a DOUBLED SQLDA */
      init_da( &sqldaPointer, sqlda_d * 2 ) ;

   /* need to reassign the SQLDA with the correct number
      of columns to the SQL statement */
   EXEC SQL DESCRIBE statement1 INTO :*sqldaPointer ;
   EMB_SQL_CHECK( "DESCRIBE" ) ;

   /* allocating the proper amount of memory
      space needed for the variables */
   alloc_host_vars( sqldaPointer ) ;  14

   /* Don't need to check the SQLCODE for declaration of cursors */
   EXEC SQL DECLARE pcurs CURSOR FOR statement1 ;  15

   EXEC SQL OPEN pcurs ;  15
   EMB_SQL_CHECK( "OPEN" ) ;

   EXEC SQL FETCH pcurs USING DESCRIPTOR :*sqldaPointer;  16
   EMB_SQL_CHECK( "FETCH" ) ;

   /* if the FETCH is successful, obtain data from SQLDA */
   /* display the column titles */
   display_col_titles( sqldaPointer ) ;  17

   /* display the rows that are fetched */
   while ( SQLCODE == 0 ) {
      counter++ ;
      display_da( sqldaPointer ) ;  18
      EXEC SQL FETCH pcurs USING DESCRIPTOR :*sqldaPointer ;
   }  /* endwhile */

   EXEC SQL CLOSE pcurs ;  19
   EMB_SQL_CHECK( "CLOSE CURSOR" ) ;
   printf( "\n %d record(s) selected\n\n", counter ) ;

   /* Free the memory allocated to this SQLDA. */
   free_da( sqldaPointer ) ;
```

```
   } else { /* this is not a SELECT statement, execute SQL statement */  13
      EXEC SQL EXECUTE statement1 ;
      EMB_SQL_CHECK( "Executing the SQL statement" ) ;
   }  /* end if */

   return( 0 ) ;

}     /* end of program : ADHOC.SQC */
```

동적 SQL로의 변수 입력

이 절에서는 사용자의 동적 SQL 응용프로그램에서 매개변수 표시문자를 사용하여 호스트 변수 정보를 표시하는 방법을 보여 줍니다. 다음 항목들이 포함됩니다.

- 매개변수 표시문자 사용
- 예: VARINP 프로그램

매개변수 표시문자 사용

호스트 변수 정보(데이터 유형 및 길이)는 응용프로그램 사전 처리 컴파일 중에만 사용가능하므로 동적 SQL문에는 호스트 변수가 포함될 수 없습니다. 실행시 호스트 변수 정보는 없습니다. 따라서, 응용프로그램 변수를 표시하려면 새로운 방법이 필요합니다. 호스트 변수는 *매개변수 표시문자*라고 하는 의문부호(?)에 의해 표시됩니다. 매개변수 표시문자에는 SQL문 내부에 호스트 변수가 위치하는 장소를 나타냅니다. 매개변수 표시문자는 SQL문 내부에서의 사용 문맥에 종속적인 데이터 유형 및 길이를 취합니다.

매개변수 표시문자의 데이터 유형이 사용되는 명령문 문맥과 구분이 되지 않는 경우 CAST를 사용하여 유형을 지정할 수 있습니다. 그런 매개변수 표시문자가 *입력된 매개변수 표시문자*로 여겨집니다. 입력된 매개변수 표시문자는 제공된 유형의 호스트 변수처럼 취급됩니다. 예를 들어, `SELECT ? FROM SYSCAT.TABLES`문은 DB2가 결과 컬럼의 유형을 알지 못하므로 유효하지 않습니다. 그러나, 유형변환 함수는 매개변수 표시문자가 INTEGER를 표시하도록 약속하므로 `SELECT CAST(? AS INTEGER) FROM SYSCAT.TABLES`문이 유효합니다. 따라서 DB2는 결과 컬럼의 유형을 압니다.

매개변수 표시문자가 포함된 문자열은 다음과 같습니다.

```
DELETE FROM TEMPL WHERE EMPNO = ?
```

이 명령문이 실행될 때 호스트 변수 또는 SQLDA 구조는 EXECUTE문의 USING 절에 의해 지정됩니다. 호스트 변수의 내용은 명령문이 실행될 때 사용됩니다.

SQL문에 하나 이상의 매개변수 표시문자가 포함될 수 있으면, EXECUTE문의 USING절은 (각 매개변수 표시문자에 대해) 호스트 변수 목록을 지정하거나 각 매개변수 표시문자에 대해 SQLVAR 항목을 가진 SQLDA를 식별해야 합니다.(LOB의 경우 매개변수 표시문자당 두 개의 SQLVAR이 있습니다.) 호스트 변수 목록 또는 SQLVAR 항목은 명령문의 매개변수 표시문자 순서에 일치하며, 호환가능한 데이터 유형이 있어야 합니다.

동적 SQL에서 매개변수 표시문자를 사용하는 것은 정적 SQL에서 호스트 변수를 사용하는 것과 같습니다. 어떤 경우이든, 최적화 알고리즘은 분산 통계를 사용하지 않고 최적 액세스 플랜을 선택하지 않을 수도 있습니다.

매개변수 표시문자에 적용되는 규칙은 *SQL 참조서*의 PREPARE문 아래에 나열됩니다.

예: VARINP 프로그램

다음은 검색 및 갱신 조건에서 매개변수 표시문자를 사용하는 UPDATE 예입니다. 샘플은 다음과 같은 프로그래밍 언어로 사용할 수 있습니다:

C varinp.sqc

Java Varinp.java

COBOL varinp.sqb

VARINP 프로그램 작업 방법

1. **SELECT SQL문 준비** PREPARE문이 SQL문을 동적으로 준비하기 위해 호출됩니다. 이 SQL문에서 매개변수 표시문자는 ?에 의해 표시됩니다. `staff`의 `job` 필드는 결과 테이블에 지정되지 않더라도 갱신이 가능하도록 정의됩니다.
2. **커서 선언.** DECLARE CURSOR문은 `c1` 커서를 **1** 에 준비된 조회에 연결시킵니다.
3. **커서 열기.** 커서 `c1`이 열립니다. 데이터베이스 관리 프로그램이 결과 테이블을 조회하고 빌드하도록 합니다. 커서는 첫번째 행 *앞에* 위치하게 됩니다.

4. **UPDATE SQL문 준비** PREPARE문이 SQL문을 동적으로 준비하기 위해 호출됩니다. 이 명령문의 매개변수 표시문자는 Clerk로 설정되지만, 갱신 중인 컬럼 데이터 유형에 일치하는 한 어떤 것으로든 동적으로 변경될 수 있습니다.
5. **행 검색.** FETCH문은 커서를 다음 행에 위치시키고 행의 내용을 호스트 변수로 이동시킵니다. 이 행은 *현재* 행이 됩니다.
6. **현재 행 갱신.** 현재 행과 지정된 컬럼 job이 전달된 매개변수 parm_var의 내용과 함께 갱신됩니다.
7. **커서 닫기.** CLOSE문이 발행됩니다. 커서와 연관된 자원들을 릴리스합니다. 하지만 커서가 또 다시 열릴 수 있습니다.

CHECKERR 매크로/함수는 오류 체크 유틸리티입니다. 이 오류 체크 유틸리티의 위치는 사용된 프로그래밍 언어에 따라 달라질 수 있습니다.

C DB2 API를 호출하는 C 프로그램에 대해서는 utilapi.c에서 sqlInfoPrint 함수는 utilapi.h에서 API_SQL_CHECK로서 참조됩니다. C embedded SQL 프로그램에 대해서는 utilemb.sqc에서 sqlInfoPrint 함수는 utilemb.h에서 EMB_SQL_CHECK로서 참조됩니다.

Java SQL 오류는 SQLException으로서 드로우되고, 응용프로그램의 catch 블록에서 처리됩니다.

COBOL CHECKERR은 checkerr.cbl라고 하는 외부 프로그램입니다.

이 오류 체크 유틸리티에 대한 소스 코드에 대해서는 137 페이지의 『예제 프로그램에서 GET ERROR MESSAGE 사용』의 내용을 참조하십시오.

C 예: VARINP.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[])
{

   EXEC SQL BEGIN DECLARE SECTION;
      char   pname[10];
      short  dept;
      char userid[9];
      char passwd[19];
      char st[255];
      char parm_var[6];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: VARINP \n" );

   if (argc == 1)
   {
      EXEC SQL CONNECT TO sample;
  EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3)
   {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else
   {
      printf ("\nUSAGE: varinp [userid passwd]\n\n");
      return 1;
   } /* endif */

   strcpy (st, "SELECT name, dept FROM staff ");
   strcat (st, "WHERE job = ? FOR UPDATE OF job");
   EXEC SQL PREPARE s1 FROM :st;  1
   EMB_SQL_CHECK("PREPARE");

   EXEC SQL DECLARE c1 CURSOR FOR s1;  2
```

```
   strcpy (parm_var, "Mgr");
   EXEC SQL OPEN c1 USING :parm_var;  3
   EMB_SQL_CHECK("OPEN");

   strcpy (parm_var, "Clerk");
   strcpy (st, "UPDATE staff SET job = ? WHERE CURRENT OF c1");
   EXEC SQL PREPARE s2 from :st;  4

   do
   {
      EXEC SQL FETCH c1 INTO :pname, :dept;  5
      if (SQLCODE != 0) break;

      printf( "%-10.10s in dept. %2d will be demoted to Clerk\n",
              pname, dept );
      EXEC SQL EXECUTE s2 USING :parm_var;  6
      EMB_SQL_CHECK("EXECUTE");
   } while ( 1 );

   EXEC SQL CLOSE c1;   7
   EMB_SQL_CHECK("CLOSE CURSOR");

   EXEC SQL ROLLBACK;
   EMB_SQL_CHECK("ROLLBACK");
   printf( "\nOn second thought -- changes rolled back.\n" );

   EXEC SQL CONNECT RESET;
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}
/* end of program : VARINP.SQC */
```

Java 예: Varinp.java

```
import java.sql.*;

class Varinp
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
    }
    catch (Exception e)
    {   System.out.println ("\n  Error loading DB2 Driver...\n");
      System.out.println (e);
      System.exit(1);
    }
  }

  public static void main(String argv[])
  {   try
    {   System.out.println ("  Java Varinp Sample");
      // Connect to Sample database

      Connection con = null;
      // URL is jdbc:db2:dbname
      String url = "jdbc:db2:sample";

      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception("\nUsage: java Varinp [username password]\n");
      }

      // Enable transactions
      con.setAutoCommit(false);

      // Perform dynamic SQL using JDBC
      try
      {   PreparedStatement pstmt1 = con.prepareStatement(
          "SELECT name, dept FROM staff WHERE job = ? FOR UPDATE OF job"); 1
      // set cursor name for the positioned update statement
      pstmt1.setCursorName("c1");  2
      pstmt1.setString(1, "Mgr");
      ResultSet rs = pstmt1.executeQuery();  3

      PreparedStatement pstmt2 = con.prepareStatement(
          "UPDATE staff SET job = ? WHERE CURRENT OF c1");  4
      pstmt2.setString(1, "Clerk");

      System.out.print("\n");
      while( rs.next() )  5
      {   String name = rs.getString("name");
        short  dept = rs.getShort("dept");
        System.out.println(name + " in dept. " + dept
          + " will be demoted to Clerk");
```

```
            pstmt2.executeUpdate();    6
          };

          rs.close();
          pstmt1.close();   7
          pstmt2.close();
          }
          catch( Exception e )
          {   throw e;
          }
          finally
          {   // Rollback the transaction
            System.out.println("\nRollback the transaction...");
            con.rollback();
            System.out.println("Rollback done.");
          }
        }
        catch( Exception e )
        {   System.out.println(e);
        }
      }
    }
```

COBOL 예: VARINP.SQB

```
 Identification Division.
 Program-ID. "varinp".

 Data Division.
 Working-Storage Section.

     copy "sqlca.cbl".

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 pname               pic x(10).
 01 dept                pic s9(4) comp-5.
 01 st                  pic x(127).
 01 parm-var            pic x(5).
 01 userid              pic x(8).
 01 passwd.
   49 passwd-length     pic s9(4) comp-5 value 0.
   49 passwd-name       pic x(18).
     EXEC SQL END DECLARE SECTION END-EXEC.

 77 errloc            pic x(80).

 Procedure Division.
 Main Section.
     display "Sample COBOL program: VARINP".

* Get database connection information.
     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

     move "SELECT name, dept FROM staff
-         "   WHERE job = ? FOR UPDATE OF job" to st.
     EXEC SQL PREPARE s1 FROM :st END-EXEC.   1
     move "PREPARE" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL DECLARE c1 CURSOR FOR s1 END-EXEC.    2

     move "Mgr" to parm-var.

     EXEC SQL OPEN c1 USING :parm-var END-EXEC    3
     move "OPEN" to errloc.
     call "checkerr" using SQLCA errloc.
```

```
     move "Clerk" to parm-var.
     move "UPDATE staff SET job = ? WHERE CURRENT OF c1" to st.

     EXEC SQL PREPARE s2 from :st END-EXEC.    4
     move "PREPARE S2" to errloc.
     call "checkerr" using SQLCA errloc.

* call the FETCH and UPDATE loop.
     perform Fetch-Loop thru End-Fetch-Loop
        until SQLCODE not equal 0.

     EXEC SQL CLOSE c1 END-EXEC.    7
     move "CLOSE" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL ROLLBACK END-EXEC.
     move "ROLLBACK" to errloc.
     call "checkerr" using SQLCA errloc.
     DISPLAY "On second thought -- changes rolled back.".

     EXEC SQL CONNECT RESET END-EXEC.
     move "CONNECT RESET" to errloc.
     call "checkerr" using SQLCA errloc.
 End-Main.
     go to End-Prog.

 Fetch-Loop Section.
     EXEC SQL FETCH c1 INTO :pname, :dept END-EXEC.    5
     if SQLCODE not equal 0
        go to End-Fetch-Loop.
     display pname, " in dept. ", dept,
        " will be demoted to Clerk".

     EXEC SQL EXECUTE s2 USING :parm-var END-EXEC.    6
     move "EXECUTE" to errloc.
     call "checkerr" using SQLCA errloc.

 End-Fetch-Loop. exit.

 End-Prog.
     stop run.
```

DB2 Call Level Interface(CLI)

Embedded SQL 인터페이스를 사용하는 응용프로그램은 SQL문을 코드로 변환하는 사전 처리 컴파일러를 필요로 하고, 이는 다시 컴파일되고, 데이터베이스로 바인드된 후 실행됩니다. 대비적으로 말해서, DB2 CLI 응용프로그램은 사전 처리 컴파일되거나 바인드될 필요가 없지만, 대신 표준 세트 함수를 사용하여 SQL문 및 관련 서비스를 수행할 때 실행합니다.

이러한 차이점은 사전 처리 컴파일러가 각 데이터베이스 제품마다 다르므로 사용자 응용프로그램을 그 제품에 효율적으로 바인드하므로 중요합니다. DB2 CLI는 특정 데이터베이스 제품과 독립적인 이식 가능한 응용프로그램을 작성할 수 있도록 합니다. 이러한 독립성은 DB2 CLI 응용프로그램을 다시 컴파일하거나 재바인드하여 DRDA 데이터베이스를 포함하여, 다른 DB2 데이터베이스에 액세스할 필요가 없습니다. 이는 런타임시 해당 데이터베이스에 연결합니다.

Embedded SQL 및 DB2 CLI 비교

DB2 CLI 및 embedded SQL은 다음과 같은 점에서도 다릅니다.

- DB2 CLI에는 명시적인 커서 선언이 필요하지 않습니다. DB2 CLI는 필요에 따라 사용되는 커서의 제공이 있습니다. 그러면 응용프로그램이 다중 행 SELECT 문 및 위치지정된 UPDATE문 및 DELETE문에 대한 일반 커서 페치 모델에서 생성된 커서를 사용할 수 있습니다.
- OPEN문은 DB2 CLI에서 사용되지 않습니다. 대신, SELECT 실행은 자동으로 커서가 열리도록 합니다.
- Embedded SQL과 달리 DB2 CLI는 EXECUTE IMMEDIATE문에 해당되는 함수(SQLExecDirect() 함수)에서 매개변수 표시문자를 허용합니다.
- DB2 CLI에서 COMMIT 또는 ROLLBACK은 SQL문으로서 전달하는 것이 아닌 SQLEndTran() 함수 호출을 통해 발행됩니다.
- DB2 CLI는 응용프로그램 대신 명령문 관련 정보를 관리하며, 이를 추상적 오브젝트로 참조하는 *명령문 핸들*을 제공합니다. 이 핸들은 응용프로그램이 제품별 데이터 구조를 사용할 필요를 줄입니다.

- 명령문 핸들과 마찬가지로 *환경* 핸들 및 *연결* 핸들은 모든 전역 변수 및 연결별 정보를 참조하는 방법을 제공합니다. *설명자* 핸들은 SQL문의 매개변수 또는 결과 세트의 컬럼에 대해 설명합니다.
- DB2 CLI는 X/Open SQL CAE 스펙에 의해 정의된 SQLSTATE 값을 사용합니다. 형식 및 대부분의 값들이 IBM 관계형 데이터베이스 제품이 사용하는 값과 일치하지 않아도 차이점이 있습니다.(ODBC SQLSTATES와 X/Open 정의 SQLSTATES 사이에도 차이가 있습니다.)
- DB2 CLI는 화면 이동 커서를 지원합니다. 화면 이동 커서를 이용하여 정적 커서로 다음과 같이 화면 이동할 수 있습니다.
  - 한 행 이상 앞으로 이동
  - 한 행 이상 뒤로 이동
  - 첫번째 행에서 한 행 이상
  - 마지막 행에서 한 행 이상

이러한 차이에도 불구하고 Embedded SQL과 DB2 CLI 사이에 중요한 공통 개념이 있습니다. *DB2 CLI는 Embedded SQL에서 동적으로 준비될 수 있는 SQL 문을 실행할 수 있습니다.*

주: DB2 CLI는 복합 SQL문과 같이 동적으로 준비될 수 없는 일부 SQL문도 허용할 수 있습니다.

829 페이지의 표37에는 각 SQL문이 나열되어 잇고, DB2 CLI를 사용하여 실행될 수 있는지 표시합니다. 또한 이 테이블은 명령행 처리기을 사용하여 (SQL문 프로토타입에 유용한) 대화식으로 명령문을 실행할 수 있는지 표시합니다.

각 DBMS에는 동적으로 준비할 수 있는 추가 명령문이 있을 수 있습니다. 이런 경우 DB2 CLI는 명령문을 DBMS로 전달합니다. 한 가지 예외가 있습니다. COMMIT문 및 ROLLBACK문은 일부 DBMS에 의해 동적으로 준비될 수 있지만 전달되지는 않습니다. 이런 경우 `SQLEndTran()` 함수를 사용하여 COMMIT 문 또는 ROLLBACK문을 지정하십시오.

DB2 CLI 사용시 이점

DB2 CLI 인터페이스에는 Embedded SQL을 통해 여러 개의 주요 이점이 있습니다.

- 응용프로그램이 작성될 때 목표 데이터베이스가 알려지지 않은 클라이언트 서버 환경에 이상적입니다. 어떤 데이터베이스 서버에 응용프로그램이 연결되어 있는지에 관계없이 SQL문 실행용으로 일치하는 인터페이스를 제공합니다.
- 사전 처리 컴파일러에 대한 종속성을 제거함으로써 응용프로그램의 이식성을 증가시킵니다.
- 개별 DB2 CLI 응용프로그램은 각 데이터베이스에 바인드될 필요가 없습니다. DB2 CLI와 함께 제공된 바인드 파일만 모든 DB2 CLI 응용프로그램에 대해 한 번 바인드되어야 합니다. 이는 일반용일 경우 응용프로그램에 필요한 관리 용량을 대폭 절감시킬 수 있습니다.
- DB2 CLI 응용프로그램은 동일한 응용프로그램으로부터 모두, 동일한 데이터베이스로의 다중 연결을 포함하여 여러 데이터베이스로 연결할 수 있습니다. 각 연결마다 고유의 확약 범위가 있습니다. 응용프로그램이 동일한 결과를 얻기 위해 멀티스레딩을 사용해야 하는 Embedded SQL을 사용하는 것보다 CLI를 사용하는 것이 훨씬 더 간단합니다.
- DB2 CLI는 제어되는 응용프로그램, 일반적으로 Embedded SQL 응용프로그램과 연결된 SQLDA 및 SQLCA와 같은 복합 데이터 영역에 대한 필요를 줄여 줍니다. 대신, DB2 CLI는 필요한 데이터 구조를 할당 및 제어하고, 응용프로그램이 참조할 *핸들*을 제공합니다.
- DB2 CLI는 각 스레드가 자신의 연결을 가질 수 있고 나머지와 분리된 확약 범위를 가질 수 있는 멀티스레드 스레드 안전 응용프로그램의 개발을 작동가능하게 합니다. DB2 CLI는 위에서 설명한 데이터 영역을 감소시키고 특정 핸들과 응용프로그램이 액세스할 수 있는 모든 데이터 구조를 연결시켜 이를 수행합니다. Embedded SQL과 달리 멀티스레드 CLI 응용프로그램은 문맥 관리 DB2 API를 호출할 필요가 없습니다. 이것은 DB2 CLI 드라이버에 의해 자동으로 처리됩니다.
- DB2 CLI는 확장된 매개변수 입력 및 페치 기능을 제공함으로써, 입력시 데이터 배열이 지정될 수 있도록 하고, 여러 행의 결과 세트를 직접 배열로 검색하고, 여러 결과 세트를 생성하는 명령문을 실행합니다.

- DB2 CLI는 다양한 DBMS 카탈로그 테이블에 포함된 조회 카탈로그(테이블, 컬럼, 외부 키, 기본 키 등) 정보에 대한 일관된 인터페이스를 제공합니다. 리턴되는 결과 세트는 DBMS를 통해 일치합니다. 그 결과 응용프로그램은 다른 데이터베이스 서버에서의 카탈로그 차이점 뿐만 아니라 데이터베이스 서버 릴리스에서의 카탈로그 변경이 이루어지지 못하도록 합니다. 따라서 응용프로그램은 버전별 및 서버별 카탈로그 조회 작성을 하지 않아도 됩니다.
- 확장 데이터 변환은 DB2 CLI에서도 제공됩니다. 다양한 SQL 및 C 데이터 유형 사이의 정보 변환시 응용프로그램 코드가 덜 필요합니다.
- DB2 CLI는 ODBC 및 X/Open CLI 함수 모두를 통합합니다. 둘다 산업 스펙을 따릅니다. DB2 CLI는 새로 도입되는 ISO CLI 표준과 함께 정렬됩니다. 응용프로그램 개발자가 이러한 스펙에 투자하는 지식은 직접 DB2 CLI 개발에 적용되거나 그 반대로 적용됩니다. 이 인터페이스는 함수 라이브러리에 대해 잘 알고 있지만 SQL문을 호스트 언어로 내포시키는 제품별 메소드에 대한 지식은 거의 없는 프로그래머가 직관적으로 이해할 수 있게 되어 있습니다.
- DB2 CLI는 DB2 Universal Database(또는 MVS/ESA용 DB2 버전 5 이후) 서버에 상주하는 저장 프로시듀어로부터 생성된 다중 행 및 결과 세트 검색 기능을 제공합니다. 그러나, 이 기능은 저장 프로시듀어가 DataJoiner 버전 2 서버로부터 액세스가능한 서버에 상주할 경우 Embedded SQL을 사용하여 버전 5 DB2 Universal Database 클라이언트용입니다.
- DB2 CLI는 배열 출력과 함께 사용할 수 있는 서버측 스크롤가능 커서를 지원합니다. 이는 Page Up, Page Down, Home 및 End 키를 사용하는 스크롤 상자에 데이터베이스 정보를 표시하는 GUI 응용프로그램에 유용합니다. 읽기 전용 커서를 스크롤 가능으로 선언할 수 있고, 결과 세트를 통해 한 행 이상 앞뒤로 이동합니다. 또한 다음으로부터 옵셋을 지정하여 행을 페치할 수도 있습니다.
  – 현재 행
  – 결과 세트의 시작 또는 끝
  – 북마크로 이전에 설정한 특정 행.
- DB2 CLI 응용프로그램은 CLI 및 Embedded SQL 응용프로그램이 결과 세트를 설명하는 것과 같은 방식으로 SQL문에 있는 동적으로 매개변수를 설명할 수 있습니다. 이를 통해 CLI 응용프로그램은 미리 이러한 매개변수 표시문자의

데이터 유형을 알지 못한 채 매개변수 표시문자가 포함된 SQL문을 동적으로 처리할 수 있도록 합니다. SQL문이 준비될 때 매개변수의 데이터 유형을 상세하게 설명하는 정보가 리턴됩니다.

Embedded SQL 또는 DB2 CLI에 대한 결정

선택할 인터페이스는 사용자 응용프로그램에 따라 다릅니다.

DB2 CLI는 이식성이 필요한 조회에 기반을 둔 그래픽 사용자 인터페이스(GUI)에 이상적입니다. 위에 나열된 DB2 CLI 사용시 이점들로 인해 응용프로그램이 확실하게 선택할 수 있는 항목이 될 수 있습니다. 그러나 한 가지 반드시 고려해야 할 사항이 있는데, 바로 정적 SQL과 동적 SQL 사이의 비교입니다. 내포된 응용프로그램에서는 정적 SQL을 사용하는 것이 훨씬 쉽습니다.

CLI 응용프로그램에서 정적 SQL을 사용하는 방법에 대해서는 다음 웹 페이지를 참조하십시오.

```
http://www.ibm.com/software/data/db2/udb/staticcli
```

정적 SQL에는 다음과 같은 이점이 있습니다.

- 성능

  동적 SQL은 런타임시 준비되고, 정적 SQL은 사전 처리 컴파일시 준비됩니다. 보다 나은 처리가 필요할 분만 아니라 준비 단계는 런타임시 추가 네트워크 통신량을 야기시킬 수 있습니다. 그러나 이러한 추가 단계(및 네트워크 통신)는 DB2 CLI 응용프로그램이 지연된 준비를 사용할 경우 필요하지 않습니다.

  정적 SQL이 동적 SQL보다 항상 성능이 나은 것은 아닙니다. 동적 SQL은 새 색인과 같은 데이터베이스로의 변경을 사용할 수 있고, 현재 데이터베이스 통계를 사용하여 최적의 액세스 플랜을 선택할 수 있습니다. 또한, 명령문이 캐쉬될 경우 사전 처리 컴파일을 피할 수 있습니다.

- 캡슐화 및 보안

  정적 SQL에서 (테이블, 뷰와 같은) 오브젝트에 대한 권한 부여가 패키지와 연관되어 있고 패키지 바인딩시 검증됩니다. 이는 데이터베이스 관리자가 각 데이터베이스 오브젝트에 대한 명시적인 액세스를 권한 부여할 필요 없이, 특정 패키지에 대한 실행을 사용자 세트에게 부여(함으로써 패키지에서 그들의 특권을 캡슐화)하기만 하면 됨을 의미합니다. 동적 SQL에서 권한 부여는 명령문 각각

에 대해 런타임시 검증됩니다. 따라서 사용자에게는 각 데이터베이스 오브젝트 에 대한 명시적인 액세스가 부여되어야 합니다. 이로써 이 사용자들은 액세스 할 필요가 없는 오브젝트 부분들에 액세스할 수 있습니다.

- Embedded SQL은 C 또는 C++가 아닌 언어로 지원됩니다.
- 고정 조회 선택의 경우 Embedded SQL은 더 간단합니다.

응용프로그램에 두 인터페이스 모두의 이점이 필요한 경우, 정적 SQL이 포함된 저장 프로시듀어를 작성하여 DB2 CLI 응용프로그램 내에 정적 SQL을 사용할 수 있습니다. 저장 프로시듀어는 DB2 CLI 응용프로그램 내부로부터 호출되고 서버에서 실행됩니다. 일단 저장 프로시듀어가 작성되면 DB2 CLI 또는 ODBC 응용프로그램이 이를 호출할 수 있습니다. *CLI Guide and Reference*에서 자세한 내용을 참조하십시오.

CLI 응용프로그램에서 정적 SQL을 사용하는 방법에 대해서는 다음 웹 페이지를 참조하십시오.

```
http://www.ibm.com/software/data/db2/udb/staticcli
```

또한, 각각의 장점을 이용하여 DB2 CLI 및 embedded SQL 모두를 사용하는 혼합 응용프로그램을 작성하는 것도 가능합니다. 이런 경우 DB2 CLI를 사용하여 기본 응용프로그램을 제공할 수 있고, 이 때 성능 또는 보안상의 이유로 정적 SQL을 사용하여 키 모듈을 이용할 수 있습니다. 이것은 응용프로그램 설계를 복잡하게 하고, 저장 프로시듀어가 응용프로그램 요건을 충족시키지 못하는 경우에만 사용되어야 합니다. 자세한 내용은 *CLI Guide and Reference*에 있는 *Embedded SQL 및 DB2 CLI 혼합* 절을 참조하십시오.

결과적으로, 각 인터페이스 사용 시기에 대한 결정은 어떤 한 요인이 아닌, 개인적인 선호사항이나 이전 경험에 기초할 것입니다.

제6장 공통 DB2 응용프로그램 기술

컬럼 생성 196
식별 컬럼 196
선언된 임시 테이블 197
저장점이 있는 트랜잭션 제어 200
저장점 제한사항 201
저장점 및 DDL(Data Definition Language) 202
저장점 및 버퍼 삽입. 203
커서 블로킹과 함께 저장점 사용. . . . 203
저장점 및 XA 호환 트랜잭션 관리 프로그램 204

DB2에서 embedded SQL을 사용하여 공통 데이터베이스 응용프로그램 개발 문제를 처리할 수 있습니다.

컬럼 생성

불완전한 삽입 및 갱신 트리거를 사용하는 대신 DB2에서는 GENERATED ALWAYS AS절을 사용하여 테이블에 생성된 컬럼을 포함시킬 수 있습니다. 생성된 컬럼은 SQL 표현식에서 나온 자동으로 갱신된 값을 제공합니다.

식별 컬럼

DB2 응용프로그램 개발자는 종종 테이블의 모든 행에 대해 기본 키를 작성해야 합니다. 기본 키로 식별 컬럼을 사용하는 테이블을 작성하면 DB2는 자동으로 고유 값을 삽입합니다. 식별 컬럼을 사용하면 응용프로그램은 잠금 경합의 감소로 향상된 성능을 얻을 수 있습니다.

선언된 임시 테이블

선언된 임시 테이블은 일반 테이블과 비슷하지만, 데이터베이스 연결 시간만큼만 지속하고 잠금 또는 로그의 대상이 아닙니다. 응용프로그램에서 많은 양의 데이터를 처리하는 테이블을 작성하고 일단 그 데이터를 처리하고 나서 테이블을 제거하면 선언된 임시 테이블 사용에 대해 고려해 보십시오. 선언된 임시 테이블은 응용프로그램의 성능을 향상시키고 동시 사용자에 대해 설계된 응용프로그램에 대해서는 더 쉽게 응용프로그램을 개발할 수 있게 합니다.

© Copyright IBM Corp. 1993, 2000

외부 저장점

COMMIT 및 ROLLBACK 문을 통해 전체 트랜잭션의 동작을 제어할 수 있는 반면, 저장점을 사용하면 트랜잭션에서 좀 더 집중적으로 제어할 수 있습니다. 저장점 블록은 여러 SQL문을 그룹을 만듭니다. 저장점 블록에 있는 하위 명령문 중 하나에서 오류가 발생하면, 오류가 발생한 하위 명령문만 구간 복원하고 기타 하위 명려문의 작업을 완료합니다.

컬럼 생성

생성된 컬럼은 삽입 또는 갱신 조작 대신 표현식에서 각 행의 값을 가져오는 컬럼입니다. 갱신 트리거와 삽입 트리거를 조합하면 비슷한 효과를 얻을 수 있는 반면, 생성된 컬럼을 사용하면 파생된 값이 표현식과 일치하게 만들 수 있습니다.

테이블에서 생성된 컬럼을 작성하려면 컬럼에 대해 GENERATED ALWAYS AS 절을 사용하고 컬럼의 값이 파생되는 표현식을 포함하십시오. GENERATED ALWAYS AS절을 ALTER TABLE 또는 CREATE TABLE문에 포함시킬 수 있습니다. 다음 예는 "c1" 및 "c2"의 일반 컬럼과 테이블의 일반 컬럼에서 파생된 "c3" 및 "c4"의 생성된 컬럼의 테이블을 작성합니다.

```
CREATE TABLE T1(c1 INT, c2 DOUBLE,
                c3 DOUBLE GENERATED ALWAYS AS (c1 + c2),
                c4 GENERATED ALWAYS AS
                  (CASE
                     WHEN c1 > c2 THEN 1
                     ELSE NULL
                   END)
               );
```

생성된 컬럼을 사용하여 응용프로그램의 성능을 개선하는 방법에 대한 자세한 내용은 *관리 안내서*을 참조하십시오. 생성된 컬럼 작성 방법에 대한 자세한 내용은 *SQL 참조서*에 있는 CREATE TABLE 문 구문을 참조하십시오.

식별 컬럼

식별 컬럼은 테이블에 있는 모든 행에 고유 기본 키 값을 자동으로 생성하는 간단한 방법을 DB2 응용프로그램 개발자에게 제공합니다. 식별 컬럼을 작성하려면 CREATE TABLE 또는 ALTER TABLE문에 IDENTITY절을 포함시키십시오.

응용프로그램에서 식별 컬럼을 사용하여 동시성과 응용프로그램이 데이터베이스 외부에서 고유 카운터를 생성할 때 발생하는 성능 문제를 피하십시오. 식별 컬럼을 사용하여 고유 기본 키를 생성하지 않는 경우에는 행이 하나인 테이블에 카운터를 저장하는 것이 일반적인 설계입니다. 각 트랜잭션은 이 테이블을 잠그고 숫자를 증가시킨 후 트랜잭션을 확약하여 카운터를 잠금해제합니다. 불행히도, 이 설계에서는 단일 트랜잭션만 카운터를 한번 증가시킬 수 있습니다.

이와 달리 식별 컬럼을 사용하여 기본 키를 자동으로 생성하는 경우, 응용프로그램은 훨씬 높은 동시성 레벨을 얻을 수 있습니다. 식별 컬럼을 사용하면 트랜잭션이 카운터를 잠그지 않아도 되도록 DB2는 카운터를 유지보수합니다. 카운터를 증가시킨 확약되지 않은 트랜잭션은 기타 후속 트랜잭션이 카운터를 증가시키지 못하도록 막지 못하므로 식별 컬럼을 사용하는 응용프로그램은 더 나은 성능을 얻을 수 있습니다.

식별 컬럼의 카운터는 트랜잭션과는 무관하게 증가되거나 감소됩니다. 주어진 트랜잭션이 식별 컬럼을 두 번 증가시키면 동일한 식별 컬럼을 동시에 증가시키는 다른 트랜잭션이 있을 수 있으므로, 그 트랜잭션은 생성된 두 숫자간에 갭이 있음을 알 수 있습니다.

캐쉬된 모든 값이 할당되기 전에 데이터베이스는 비활성화된(정상 또는 비정상) 값의 범위를 캐쉬했으므로 식별 컬럼은 구간 복원된 트랜잭션의 결과로 카운터에 갭을 생성하는 것처럼 보일 수 있습니다.

식별 컬럼에 대한 자세한 내용은 *관리 안내서*를 참조하십시오. CREATE TABLE 및 ALTER TABLE문의 IDENTITY절에 대한 자세한 정보는 *SQL 참조서*를 참조하십시오.

선언된 임시 테이블

*선언된 임시 테이블*은 임시 테이블을 작성한 응용프로그램에서 실행되는 SQL문에서만 액세스될 수 있는 임시 테이블입니다. 선언된 임시 테이블은 데이터베이스에 응용프로그램이 연결되어 있는 지속기간 이후까지 지속되지 않습니다.

선언된 임시 테이블을 사용하여 응용프로그램의 성능을 잠재적으로 향상시키십시오. 선언된 임시 테이블을 작성할 때 DB2는 시스템 카탈로그 테이블에 항목을 삽

입하지 않으므로, 서버에서 카탈로그 경합 문제가 발생하지 않습니다. 일반 테이블과 비교했을 때 DB2는 선언된 임시 테이블이나 행을 잠그지 않고 선언된 임시 테이블이나 그 내용을 기록하지 않습니다. 현재 응용프로그램에서 테이블을 작성하여 많은 양의 데이터를 처리하고, 응용프로그램에서 데이터 처리가 끝났을 때 이런 테이블을 제거하면 일반 테이블 대신 선언된 임시 테이블 사용을 고려해 보십시오.

동시 사용자에 대해 작성된 응용프로그램을 개발하면 응용프로그램은 선언된 임시 테이블을 사용합니다. 일반 테이블과는 달리 선언된 임시 테이블에서는 이름 충돌이 일어나지 않습니다. 응용프로그램의 각 인스턴스에 대해 DB2는 동일한 이름으로 선언된 임시 테이블을 작성할 수 있습니다. 예를 들어, 일반 테이블을 사용하여 많은 양의 임시 데이터를 처리하는 동시 사용자에 대해 응용프로그램을 작성하려면, 응용프로그램의 각 인스턴스에서 임시 데이터를 보유하는 일반 테이블에 고유 이름을 사용해야 합니다. 일반적으로, 언제든지 사용할 수 있는 테이블을 이름을 추적하는 다른 테이블을 작성합니다. 선언된 임시 테이블을 사용하면 임시 데이터에 대해 선언된 임시 테이블 이름을 한 번만 지정하면 됩니다. DB2는 응용프로그램의 각 인스턴스에서 고유 테이블을 사용하게 만듭니다.

선언된 임시 테이블을 사용하려면 다음 단계를 수행하십시오.

단계 1. USER TEMPORARY TABLESPACE가 있어야 합니다. USER TEMPORARY TABLESPACE가 없으면 CREATE USER TEMPORARY TABLESPACE문을 실행하십시오.

단계 2. 응용프로그램에서 DECLARE GLOBAL TEMPORARY TABLE문을 실행하십시오.

선언된 임시 테이블의 스키마는 항상 SESSION입니다. SQL문에서 선언된 임시 테이블을 사용하려면, SESSION 스키마 규정자를 명시적으로 사용하여 테이블을 참조하거나 SESSION의 DEFAULT 스키마를 사용하여 규정되지 않은 참조를 규정합니다. 다음 예에서 테이블 이름은 다음 명령문을 사용하여 TT1으로 명명된 선언된 임시 테이블을 작성할 때 SESSION 스키마 이름으로 항상 규정됩니다.

```
DECLARE GLOBAL TEMPORARY TABLE TT1
```

앞의 예에서 작성된 선언된 임시 테이블에서 *column1* 컬럼의 내용을 선택하려면 다음 명령문을 사용하십시오.

```
SELECT column1 FROM SESSION.TT1;
```

DB2를 통해 SESSION 스키마가 있는 지속 테이블을 작성할 수도 있습니다. SESSION.TT3의 완전한 이름의 지속 테이블을 작성하면, SESSION.TT3의 완전한 이름의 선언된 임시 테이블을 작성할 수 있습니다. 이런 상황에서 DB2는 항상 똑같이 규정된 이름을 사용하는 지속 및 선언된 임시 테이블에 대한 참조를 선언된 임시 테이블에서 해결합니다. 지속 테이블과 선언된 임시 테이블간의 혼란을 피하려면 SESSION 스키마를 사용하여 지속 테이블을 작성하지 말아야 합니다.

테이블, 뷰 또는 SESSI 스키마로 규정된 별명에 대한 정적 SQL 참조를 포함하는 응용프로그램을 작성하면, DB2 사전 처리 컴파일러는 바인드 시간에 그 명령문을 컴파일하지 않고 명령문을 "컴파일해야 하는 것"으로 표시합니다. 런타임시, DB2는 명령문을 컴파일합니다. 이 동작은 *증분식 바인딩*이라고 합니다. DB2는 테이블, 뷰 그리고 SESSION 스키마로 규정된 별명에 대한 정적 SQL 참조에 대해 증분식 바인딩을 자동으로 수행합니다. BIND 또는 PRECOMPILE 명령에서 VALIDATE RUN 옵션을 지정하여 이런 명령문에서 증분식 바인딩을 사용하지 않아도 됩니다.

DECLARE GLOBAL TEMPORARY TABLE문을 포함하는 트랜잭션에 대해 ROLLBACK문을 실행하면 DB2는 선언된 임시 테이블을 제거합니다. 선언된 임시 테이블에 대해 DROP TABLE문을 실행하면, 그 트랜잭션에 대해 ROLLBACK문을 실행할 때만 빈 선언된 임시 테이블을 복원할 수 있습니다. DROP TABLE문의 ROLLBACK은 선언된 임시 테이블에 있는 행을 복원하지 않습니다.

선언된 임시 테이블의 기본 동작은 트랜잭션을 확약할 때 테이블에서 모든 행을 삭제하는 것입니다. 그러나 하나 이상의 WITH HOLD 커서가 선언된 임시 테이블에 계속 열려 있으면, DB2는 트랜잭션을 확약할 때 테이블에서 행을 삭제하지 않습니다. 트랜잭션을 확약할 때 모든 행을 삭제하지 않으려면, DECLARE GLOBAL TEMPORARY TABLE문에서 ON COMMIT PRESERVE ROWS 절을 사용하여 임시 테이블을 작성합니다.

트랜잭션에서 INSERT, UPDATE 또는 DELETE문을 사용하여 선언된 임시 테이블의 내용을 수정하고, 그 트랜잭션을 구간 복원하면 DB2는 선언된 임시 테이

블의 모든 행을 삭제합니다. INSERT, UPDATE 또는 DELETE문을 사용하여 선언된 임시 테이블의 내용을 수정하려고 하는데, 명령문이 실패하면 DB2는 선언된 임시 테이블의 모든 행을 삭제합니다.

파티션된 환경에서 노드 실패가 발생하면 실패한 노드에 파티션이 있는 모든 선언된 임시 테이블은 사용할 수 없게 됩니다. 이렇게 사용할 수 없는 선언된 임시 테이블에 대한 모든 후속 액세스는 오류를 리턴합니다(SQL1477N). 응용프로그램에서 사용할 수 없는 선언된 임시 테이블을 만나면, 응용프로그램은 테이블을 삭제하거나 DECLARE GLOBAL TEMPORARY TABLE문에서 WITH REPLACE 절을 지정하여 테이블을 다시 작성할 수 있습니다.

선언된 임시 테이블은 몇 가지 제한사항의 적용을 받습니다. 예를 들어, 선언된 임시 테이블에 대해 색인, 별명 또는 뷰를 정의할 수 없습니다. IMPORT와 LOAD를 사용하여 선언된 임시 테이블에 상주시킬 수 없습니다. DECLARE GLOBAL TEMPORARY TABLE문의 완전한 구문과 선언된 임시 테이블에 대한 완전한 제한사항 목록에 대해서는 *SQL 참조서*를 참조하십시오.

저장점이 있는 트랜잭션 제어

저장점은 데이터베이스 요청이 실패할 때 DBMS에 의해 작업을 실행 취소하는 메카니즘입니다. 저장점은 비원자적 데이터베이스 요청을 원자적으로 동작하게 만듭니다. 실행 중에 오류가 발생하면 저장점을 사용하여 저장점이 시작된 시간과 저장점 구간 복원이 요청된 시간 사이에 트랜잭션에 의해 변경된 사항을 실행 취소할 수 있습니다.

저장점은 복합 SQL문과 비슷합니다. 이를 통해 여러 SQL문을 단일 실행 가능 블록으로 그룹화할 수 있습니다. 블록의 첫번째 부속 명령문이 실행되기 전에 저장점 블록을 시작하는 저장점 요청이 필요합니다. 부속 명령문 중에서 오류가 발생하면 그 부속 명령문만 구간 복원됩니다. 이는 단일 오류로 전체 블록에 오류가 발생하고 전체 복합 SQL문을 구간 복원하게 만드는 복합 SQL문보다 좀 더 집중성을 제공합니다. 명령문의 저장점 블록 끝에서 저장점을 해제하거나 저장점으로 구간 복원할 수 있습니다.

다음 SQL문으로 저장점을 작성하고 제어할 수 있습니다.

SAVEPOINT

저장점을 설정하려면 SAVEPOINT SQL문을 실행하십시오. 코드의 확실성을 향상시키려면 저장점에 의미있는 이름을 지정할 수 있습니다. 예를 들어 다음과 같습니다.

```
SAVEPOINT savepoint1 ON ROLLBACK RETAIN CURSORS
```

RELEASE SAVEPOINT

저장점을 해제하려면 RELEASE SAVEPOINT SQL문을 실행하십시오. RELEASE SAVEPOINT SQL문을 사용하여 저장점을 명시적으로 해제하지 않으면, 이는 트랜잭션 끝에서 해제됩니다. 예를 들어 다음과 같습니다.

```
RELEASE SAVEPOINT savepoint1
```

ROLLBACK TO SAVEPOINT

저장점으로 구간 복원하려면 ROLLBACK TO SAVEPOINT SQL문을 실행하십시오. 예를 들어 다음과 같습니다.

```
ROLLBACK TO SAVEPOINT
```

SAVEPOINT, RELEASE SAVEPOINT 그리고 ROLLBACK TO SAVEPOINT 문의 완전한 구문에 대해서는 *SQL 참조서*를 참조하십시오.

저장점 제한사항

DB2 Universal Database는 다음 제한사항을 응용프로그램에서 저장점을 사용할 때 적용합니다.

최소단위 복합 SQL

DB2를 통해 최소단위 복합 SQL에서 저장점을 사용할 수 없습니다. 저장점에서 최소단위 복합 SQL을 사용할 수 없습니다.

중첩 저장점

DB2는 다른 저장점에서의 저장점 사용을 지원하지 않습니다.

트리거 DB2는 트리거에서의 저장점 사용을 지원하지 않습니다.

트랜잭션에서의 저장점 수량

DB2를 통해 트랜잭션에서 필요한 만큼의 저장점을 설정하고 사용할 수 있습니다.

SET INTEGRITY문

저장점에서 DB2는 SET INTEGRITY문을 DDL문으로 처리합니다. 저장점에서 DLL을 사용하는 것에 대한 자세한 내용은 『저장점 및 DDL(Data Definition Language)』를 참조하십시오.

저장점 및 DDL(Data Definition Language)

DB2를 통해 저장점에 DDL문을 포함시킬 수 있습니다. 응용프로그램이 DDL문을 실행하는 저장점을 성공적으로 해제하면, 응용프로그램은 DDL에서 작성된 SQL 오브젝트를 계속 사용할 수 있습니다. 그러나, 응용프로그램에서 DDL문을 실행하는 저장점에 대해 ROLLBACK TO SAVEPOINT문을 실행하면, DB2는 이러한 DDL문의 영향을 받는 커서를 유효하지 않음으로 표시합니다.

다음 예에서 응용프로그램은 ROLLBACK TO SAVEPOINT문을 실행한 후 이전에 열린 세 개의 커서에서 페치하려고 합니다.

```
SAVEPOINT savepoint_name;
   PREPARE s1 FROM 'SELECT FROM t1';
   --issue DDL statement for t1
      ALTER TABLE t1 ADD COLUMN...
   PREPARE s2 FROM 'SELECT FROM t2';
   --issue DDL statement for t3
      ALTER TABLE t3 ADD COLUMN...
   PREPARE s3 FROM 'SELECT FROM t3';
   OPEN c1 USING s1;
   OPEN c2 USING s2;
   OPEN c3 USING s3;
ROLLBACK TO SAVEPOINT
FETCH c1; --invalid (SQLCODE -910)
FETCH c2; --successful
FETCH c3; --invalid (SQLCODE -910)
```

ROLLBACK TO SAVEPOINT문에서 DB2는 “c1”과 “c3” 커서를 유효하지 않음으로 표시합니다. 이런 커서가 종속되어 있는 SQL 오브젝트가 저장점의 DDL 문에서 처리되기 때문입니다. 그러나, 예에서 “c2” 커서를 사용하는 FETCH는 ROLLBACK TO SAVEPOINT문 다음에 성공적으로 이루어집니다.

CLOSE문을 실행하여 유효하지 않은 커서를 닫을 수 있습니다. 유효하지 않은 커서에 대해 FETCH를 실행하면 DB2는 SQLCODE –910을 리턴합니다. 유효하지 않은 커서에 대해 OPEN문을 실행하면 DB2는 SQLCODE –502을 리턴합니

다. 유효하지 않은 커서에 대해 UPDATE 또는 DELETE WHERE CURRENT OF문을 실행하면 DB2는 SQLCODE −910을 리턴합니다.

저장점에서 DB2는 NOT LOGGED INITIALLY 특성의 테이블과 임시 테이블을 다음과 같이 처리합니다.

NOT LOGGED INITIALLY 테이블
저장점에서 NOT LOGGED INITIALLY 특성이 있는 테이블을 작성하거나 NOT LOGGED INITIALLY 특성을 갖도록 테이블을 변경할 수 있습니다. 그러나 이런 저장점에 대해 DB2는 ROLLBACK TO SAVEPOINT문을 ROLLBACK WORK문으로 처리하고 전체 트랜잭션을 구간 복원합니다.

저장점 내부의 DECLARE TEMPORARY TABLE
임시 테이블이 저장점에서 선언되면 ROLLBACK TO SAVEPOINT문이 임시 테이블을 제거합니다.

저장점 외부의 DECLARE TEMPORARY TABLE
임시 테이블이 저장점 외부에서 선언되면 ROLLBACK TO SAVEPOINT 문이 임시 테이블을 제거하지 않습니다.

저장점 및 버퍼 삽입

DB2 응용프로그램의 성능을 향상시키려면 INSERT BUF 옵션을 사용하여 사전 처리 컴파일하거나 바인드하여 응용프로그램에서 버퍼 삽입을 사용할 수 있습니다. 응용프로그램에서 버퍼 삽입과 저장점을 둘다 사용하면, DB2는 SAVEPOINT, RELEASE SAVEPOINT, OR ROLLBACK TO SAVEPOINt문을 실행하기 전에 버퍼를 비웁니다.

응용프로그램에서 버퍼 삽입을 사용하는 것에 대한 자세한 내용은 623 페이지의 『버퍼 삽입 사용』을 참조하십시오. 응용프로그램 사전 처리 컴파일과 바인드에 대한 자세한 내용은 *Command Reference*를 참조하십시오.

커서 블로킹과 함께 저장점 사용

응용프로그램에서 저장점을 사용하면, 사전 처리 컴파일 옵션 BLOCKING NO를 사용하여 응용프로그램을 사전 처리 컴파일하거나 바인드하여 커서를 클럭하지 못

하도록 막는 것에 대해 고려해 보십시오. 블로킹 커서가 여러 행을 사전에 페치하여 응용프로그램의 성능을 향상시킬 수 있지만, 저장점을 사용하는 응용프로그램에서 리턴한 데이터와 블로킹 커서는 데이터베이스에 확약된 데이터를 반영하지 않습니다.

BLOCKING NO를 사용하여 응용프로그램을 사전 처리 컴파일하지 않고, 응용프로그램에서 ROLLBACK TO SAVEPOINT한 후에 FETCH문을 실행하면 FETCH문은 삭제된 데이터를 검색할 수 있습니다. 예를 들어, 다음 SQL에 들어 있는 응용프로그램이 BLOCKING NO 옵션 없이 사전 처리 컴파일된다고 가정해 보십시오.

```
CREATE TABLE t1(c1 INTEGER);
DECLARE CURSOR c1 AS 'SELECT c1 FROM t1 ORDER BY c1';
INSERT INTO t1 VALUES (1);
SAVEPOINT showFetchDelete;
   INSERT INTO t1 VALUES (2);
   INSERT INTO t1 VALUES (3);
   OPEN CURSOR c1;
   FETCH c1; --get first value and cursor block
   ALTER TABLE t1... --add constraint
ROLLBACK TO SAVEPOINT;
FETCH c1; --retrieves second value from cursor block
```

응용프로그램이 “t1” 테이블에서 첫번째 FETCH를 실행하면, DB2 서버는 컬럼 값의 블록(1, 2 및 3)을 클라이언트 응용프로그램에 보냅니다. 이런 컬럼 값은 클라이언트에 지역적으로 저장됩니다. 응용프로그램이 ROLLBACK TO SAVEPOINT SQL문을 실행하면 '2'와 '3' 컬럼 값은 테이블에서 삭제됩니다. ROLLBACK TO SAVEPOINT문 다음에 테이블의 다음 FETCH는 '2' 컬럼 값이 더 이상 테이블에 없어도 이를 리턴합니다. 응용프로그램은 커서 블로킹 옵션을 사용하여 성능을 향상시키고 지역적으로 저장된 데이터를 액세스하므로 이 값을 수신합니다.

응용프로그램 사전 처리 컴파일과 바인드에 대한 자세한 내용은 *Command Reference*를 참조하십시오.

저장점 및 XA 호환 트랜잭션 관리 프로그램

XA 호환 트랜잭션 관리 프로그램이 XA_END 요청을 실행할 때 응용프로그램에 활성화된 저장점이 있으면 DB2는 RELEASE SAVEPOINT문을 실행합니다.

제3부 저장 프로시듀어

© Copyright IBM Corp. 1993, 2000

제7장 저장 프로시듀어

저장 프로시듀어 개요 207
저장 프로시듀어의 이점. 208
저장 프로시듀어 작성 211
클라이언트 응용프로그램 213
호스트 변수 할당. 213
저장 프로시듀어 호출 214
클라이언트 응용프로그램 수행 . . . 214
서버에서의 저장 프로시듀어 214
저장 프로시듀어 레지스터 214
변수 선언 및 CREATE
PROCEDURE 예 229
저장 프로시듀어의 SQL문 231
중첩 저장 프로시듀어 232
제한사항. 233
OLE 자동화 저장 프로시듀어 작성 . . . 234
예제 OUT 매개변수 저장 프로시듀어 . . 235
OUT 클라이언트 설명 237
예제 OUT 클라이언트 응용프로그램:
Java 240
예제 OUT 클라이언트 응용프로그램:
C 242
OUT 저장 프로시듀어 설명 243
예제 OUT 매개변수 저장 프로시듀어:
Java 244
예제 OUT 매개변수 저장 프로시듀어:
C 245
코드 페이지 고려사항 247
C++ 고려사항. 248
그래픽 호스트 변수 고려사항 248
다중사이트 갱신 고려사항 249
NOT FENCED 저장 프로시듀어 249
저장 프로시듀어로부터 결과 세트 리턴. . . 251
예제: 저장 프로시듀어로부터 결과 세트
리턴 252
C 예: SPSERVER.SQC
(one_result_set_to_client) 254
Java 예: Spserver.java
(resultSetToClient) 255
예제: 저장 프로시듀어로부터 결과 세트
허용 255
문제점 해결. 260

저장 프로시듀어 개요

저장 프로시듀어를 사용하여 클라이언트/서버 응용프로그램의 성능을 향상시키십시오. *저장 프로시듀어*는 데이터베이스 서버에서 액세스할 수 있는 공유 라이브러리에 있는 함수입니다. 저장 프로시듀어는 데이터베이스를 지역적으로 액세스하고 정보를 *클라이언트 응용프로그램*에 리턴합니다. 저장 프로시듀어는 원격 응용프로그램이 복수 SQL문을 서버로 전달하도록 하는 오버헤드를 없애줍니다. 단일 CALL 문을 사용하여 클라이언트 응용프로그램은 저장 프로시듀어를 호출하고, 데이터베이스 액세스 작업을 수행하며 결과를 클라이언트 응용프로그램에 리턴합니다.

© Copyright IBM Corp. 1993, 2000

SQL을 사용하여 *SQL 프로시듀어*라고 하는 저장 프로시듀어를 작성할 수 있습니다. SQL 프로시듀어 작성에 대해서는 263 페이지의 『제8장 SQL 프로시듀어 작성』의 내용을 참조하십시오. C 또는 Java와 같은 언어를 사용하여 저장 프로시듀어를 작성할 수도 있습니다. 저장 프로시듀어와 같은 언어로 클라이언트 응용프로그램을 작성하지 않아도 됩니다. 클라이언트 응용프로그램과 저장 프로시듀어의 언어가 다르면 DB2는 클라이언트와 저장 프로시듀어간에 값을 투명하게 전달합니다.

DB2 Stored Procedure Builder(SPB)를 사용하면 Java 또는 SQL 저장 프로시듀어 개발이 간편해집니다. 사용자는 Microsoft Visual Studio 및 IBM Java용 Visual Age와 같은 대중적인 응용프로그램 개발 도구와 통합될 수 있고, 이를 독립형 유틸리티로 사용할 수도 있습니다. 사용자가 직접 저장 프로시듀어를 작성하는 것을 돕기 위해 SPB는 기본 설계 패턴을 안내하는 설계 보조 기능을 제공하고, SQL 조회 작성을 돕고, 저장 프로시듀어 호출의 성능상의 비용을 평가합니다.

DB2 Stored Procedure Builder에 대한 세부사항은 289 페이지의 『제9장 IBM DB2 Stored Procedure Builder』의 내용을 참조하십시오.

저장 프로시듀어의 이점

그림3에서는 데이터베이스 서버에 위치한 데이터베이스에 일반 데이터베이스 관리 프로그램 응용프로그램이 어떻게 액세스하는지를 보여줍니다.

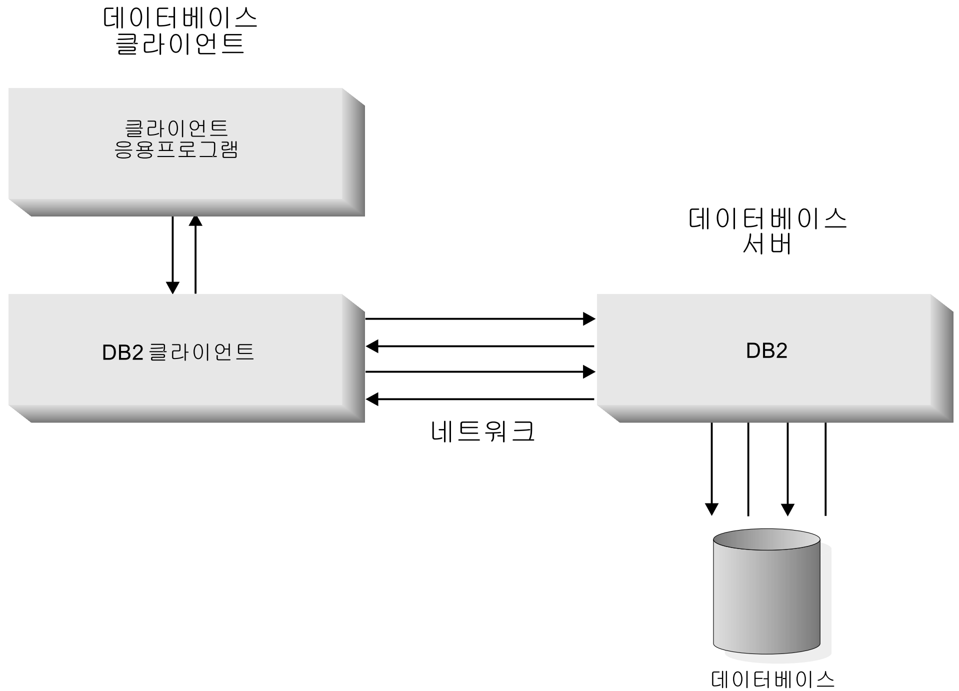


그림 3. 서버의 데이터베이스에 액세스하는 응용프로그램

모든 데이터베이스 액세스는 네트워크를 경유하므로, 성능이 나빠지는 경우도 있습니다.

저장 프로시듀어를 사용함으로써 클라이언트 응용프로그램은 데이터베이스 서버에서의 저장 프로시듀어로 제어권을 전달합니다. 이를 통해 저장 프로시듀어는 *네트워크를 통해 불필요한 데이터를 전송하지 않고* 데이터베이스 서버에서 중간 처리를 수행할 수 있습니다. 실제로 클라이언트에 필요한 레코드만 전송됩니다. 그 결과 네트워크 전송량이 감소되고 전반적으로 성능이 향상됩니다. 그림4에 이러한 기능이 나와 있습니다.

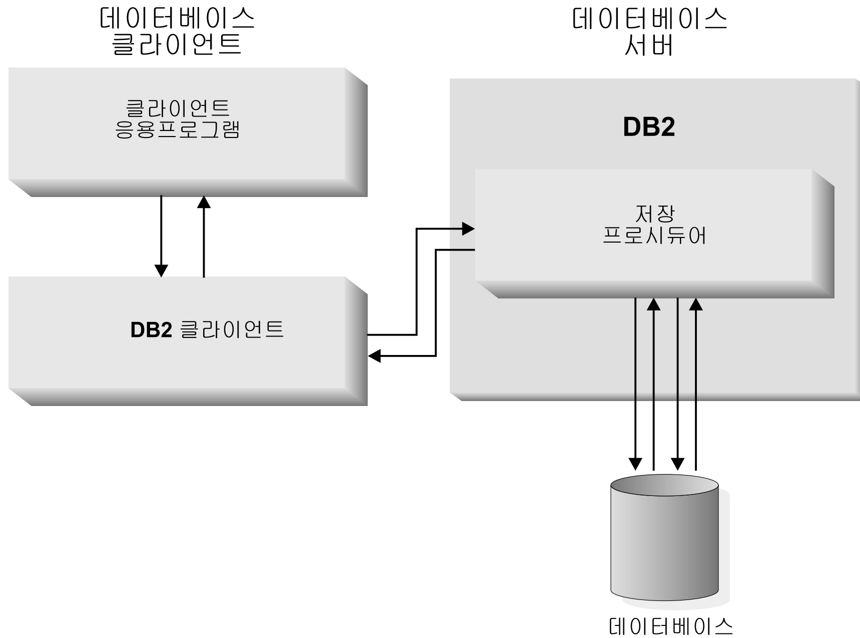


그림 4. 저장 프로시듀어 응용프로그램 사용

저장 프로시듀어를 사용하는 응용프로그램은 다음과 같은 이점이 있습니다.

- 네트워크 통신량 감소

  저장 프로시듀어를 사용하여 대량의 데이터를 처리하는 적절히 설계된 응용프로그램은 클라이언트가 필요로 하는 데이터만 리턴합니다. 따라서 네트워크를 통해 전송되는 데이터량이 감소됩니다.

- 서버 집중 작업 성능 향상

  더 많은 SQL문들이 그룹화될수록 네트워크 통신량이 그만큼 감소하게 됩니다. 일반 응용프로그램의 경우 SQL문 하나에 대해 네트워크를 두 번 경유하게 되는 반면, 저장 프로시듀어 기술을 사용하는 응용프로그램은 SQL문 그룹에 대해 네트워크 경유를 두 번 하게 됩니다. 따라서 네트워크 경유 횟수가 감소되고, 그 결과 한 번의 경유와 관련된 오버헤드가 줄어듭니다.

- 데이터베이스 서버에만 있는 기능에 대한 액세스는 다음과 같습니다.
  - (LIST DATABASE DIRECTORY 및 LIST NODE DIRECTORY와 같이) 서버 상에 디렉토리를 나열하는 명령은 서버에서만 수행될 수 있습니다.
  - 저장 프로시듀어는 서버 컴퓨터가 그렇게 설비된 경우, 메모리 및 디스크 공간을 증가시킬 수 있습니다.

– 데이터베이스 서버에만 설치된 추가 소프트웨어는 저장 프로시듀어를 통해 액세스할 수 있습니다.

저장 프로시듀어 작성

저장 프로시듀어를 포함하는 응용프로그램 설계는 별도의 클라이언트와 서버 응용프로그램으로 구성됩니다. *저장 프로시듀어*라는 서버 응용프로그램은 서버의 공유 라이브러리 또는 클래스 라이브러리에 들어 있습니다. 데이터베이스가 상주하는 서버 인스턴스에서 저장 프로시듀어를 컴파일하고 액세스해야 합니다. *클라이언트 응용프로그램*에는 저장 프로시듀어에 대한 CALL문이 있습니다. CALL문은 매개변수를 저장 프로시듀어에 전달하고 저장 프로시듀어에서 매개변수를 리턴할 수 있습니다. 다른 언어를 사용하여 저장 프로시듀어와 클라이언트 응용프로그램을 작성할 수 있습니다. 클라이언트 응용프로그램은 저장 프로시듀어와 다른 플랫폼에서 실행될 수 있습니다.

클라이언트 응용프로그램은 다음 타스크를 수행합니다.

1. 선택적 데이터 구조 및 호스트 변수에 대한 저장영역을 선언, 할당 및 초기화합니다.
2. 내재된 연결을 수행하거나 CONNECT TO문을 실행하여 데이터베이스에 연결합니다. *SQL 참조서*에서 자세한 내용을 참조하십시오.
3. SQL CALL문을 통해 저장 프로시듀어를 호출합니다.
4. 데이터베이스로 COMMIT 또는 ROLLBACK을 발행합니다.

   주: 저장 프로시듀어가 COMMIT 또는 ROLLBACK문을 실행할 수 있지만 클라이언트 응용프로그램에서 COMMIT 또는 ROLLBACK을 실행하는 것이 좋습니다. 이를 통해 클라이언트 응용프로그램은 저장 프로시듀어에 의해 리턴된 데이터를 평가하고 트랜잭션을 확약할 것인지 아니면 구간 복원할 것인지를 결정할 수 있습니다.
5. 데이터베이스로부터의 연결을 해제합니다.

위 단계를 통해 SQL문을 코드화할 수 있습니다.

저장 프로시듀어가 호출되면 다음 타스크를 수행하게 됩니다.

1. 클라이언트 응용프로그램으로부터 매개변수를 받습니다.

2. 클라이언트 응용프로그램과 같은 트랜잭션하에서 데이터베이스 서버에서 실행됩니다.
3. 선택적으로 하나 이상의 COMMIT 또는 ROLLBACK문을 실행하십시오.

   주: 저장 프로시듀어가 COMMIT 또는 ROLLBACK문을 실행할 수 있지만 클라이언트 응용프로그램에서 COMMIT 또는 ROLLBACK을 실행하는 것이 좋습니다. 이를 통해 클라이언트 응용프로그램은 저장 프로시듀어에 의해 리턴된 데이터를 평가하고 트랜잭션을 확약할 것인지 아니면 구간 복원할 것인지를 결정할 수 있습니다.
4. SQLCA 정보 및 선택적 출력 데이터를 클라이언트 응용프로그램으로 리턴합니다.

저장 프로시듀어는 클라이언트 응용프로그램에 의해 호출될 때 실행됩니다. 서버 프로시듀어가 처리를 종료할 때 클라이언트로 제어가 리턴됩니다. 여러 개의 저장 프로시듀어를 하나의 라이브러리에 둘 수 있습니다.

이 장에서는 다음과 같은 매개변수 스타일을 이용하여 저장 프로시듀어를 작성하는 방법에 대해 설명합니다.

DB2SQL
저장 프로시듀어는 클라이언트 응용프로그램의 CALL문으로부터 CREATE PROCEDURE문에서 호스트 변수로 선언되는 매개변수를 받습니다. DB2는 DB2SQL 저장 프로시듀어에 추가 매개변수를 할당합니다.

GENERAL
저장 프로시듀어는 클라이언트 응용프로그램에서 CALL문으로부터 호스트 변수로서 매개변수를 받습니다. 저장 프로시듀어는 널(NULL) 표시기를 직접 클라이언트 응용프로그램에 전달하지 않습니다. GENERAL은 OS/390용 DB2 Universal Database에 대한 SIMPLE 저장 프로시듀어와 같습니다.

GENERAL WITH NULLS
사용자가 선언하는 각 매개변수에 대해 DB2는 해당 INOUT 매개변수 널 표시기를 할당합니다. GENERAL과 마찬가지로, 매개변수는 호스트 변수로서 전달됩니다. GENERAL WITH NULLS은 OS/390용 DB2 Universal Database에 대한 SIMPLE WITH NULLS 저장 프로시듀어와 같습니다.

JAVA 저장 프로시듀어는 SQLJ 루틴 스펙과 일치하는규약을 전달하는 매개변수를 사용합니다. 저장 프로시듀어는 호스트 변수로서 IN 매개변수를 받고, 단일 항목 배열로서 OUT 및 INOUT 매개변수를 받습니다.

사용자는 반드시 CREATE PROCEDURE문을 이용하여 이전에 나열된 매개변수 스타일에 대해 각 저장 프로시듀어를 등록해야 합니다. CREATE PROCEDURE 문은 각 저장 프로시듀어의 프로시듀어 이름, 인수, 위치 및 매개변수 스타일을 지정합니다. 이러한 매개변수 스타일은 DB2 계열을 통해 저장 프로시듀어 코드의 이식성 및 확장성을 증가시킵니다.

DB2 Universal Database 버전 6 이전의 DB2 버전, 즉 DB2DARI 및 DB2GENERAL 매개변수 스타일로 지원되는 저장 프로시듀어의 스타일만 사용하는 것에 대한 내용은 857 페이지의 『부록C. DB2DARI 및 DB2GENERAL 저장 프로시듀어와 UDF』를 참조하십시오.

클라이언트 응용프로그램

클라이언트 응용프로그램은 저장 프로시듀어를 호출하기 전에 몇 가지 단계를 수행합니다. 우선 데이터베이스에 연결해야 하고, 호스트 변수나 SQLDA 구조를 선언, 할당 및 초기화해야 합니다. SQL CALL문은 일련의 호스트 변수 또는 하나의 SQLDA 구조를 수용할 수 있습니다. SQL CALL문 및 SQLDA 구조에 대한 설명은 *SQL 참조서*의 내용을 참조하십시오. 클라이언트 응용프로그램에서 SQLDA 구조를 사용하는 것에 대해서는 857 페이지의 『부록C. DB2DARI 및 DB2GENERAL 저장 프로시듀어와 UDF』의 내용을 참조하십시오.

호스트 변수 할당

저장 프로시듀어의 클라이언트측에 필요한 입력 호스트 변수를 할당하려면 다음 단계를 사용하십시오.

1. 저장 프로시듀어로 전달될 모든 입력 변수에 대해 충분한 호스트 변수를 선언하십시오.
2. 저장 프로시듀어에서 클라이언트로 값을 되돌리기 위해 어떤 입력 호스트 변수를 사용할 수 있는지도 결정하십시오.
3. 저장 프로시듀어로부터 클라이언트로 리턴되는 추가 값에 대한 호스트 변수를 선언하십시오.

저장 프로시듀어의 클라이언트 부분을 작성할 때 입력 및 출력 모두에 대해 사용함으로써 가능한 한 많은 호스트 변수를 오버로드해야 합니다. 그 결과 여러 호스트 변수를 처리하는 데 있어 효율성이 증가됩니다. 예를 들어, SQLCODE를 저장 프로시듀어로부터 클라이언트로 리턴할 때, INTEGER로서 선언되는 입력 호스트 변수를 사용하여 SQLCODE를 리턴해 보십시오.

주: 데이터베이스 서버에서 이들 구조에 대한 저장영역을 할당하지 마십시오. 데이터베이스 관리 프로그램은 클라이언트 응용프로그램에 의해 할당된 저장영역에 기초하여 자동으로 이중 저장영역을 할당합니다. 저장 프로시듀어측의 입력/출력 매개변수에 대한 저장영역 포인터를 변경하지 마십시오. 지역적으로 작성된 저장영역 포인터로 포인터를 대체하면 SQLCODE -1133 (SQLSTATE 39502) 오류가 발생할 수 있습니다.

저장 프로시듀어 호출

SQL CALL문을 사용하여 데이터베이스 위치에 저장된 저장 프로시듀어를 호출할 수 있습니다. CALL문에 대한 완전한 설명은 *SQL 참조서*의 내용을 참조하십시오. 저장 프로시듀어를 호출할 때 CALL문을 사용하는 것이 좋습니다.

클라이언트 응용프로그램 수행

클라이언트 응용프로그램은 저장 프로시듀어를 호출하기 전에 데이터베이스 연결을 해야 합니다. 그렇지 않으면 오류가 리턴됩니다. 데이터베이스 연결 및 데이터 구조가 초기화된 후, 클라이언트 응용프로그램은 저장 프로시듀어를 호출하고 필요한 데이터를 전달합니다. 응용프로그램은 데이터베이스로부터 연결해제됩니다. 위 단계를 통해 SQL문을 코드화할 수 있습니다.

서버에서의 저장 프로시듀어

저장 프로시듀어는 SQL CALL문에 의해 호출되며, 클라이언트 응용프로그램에 의해 전달된 데이터를 사용합니다. CREATE PROCEDURE문을 이용하여 데이터베이스 관리 프로그램에 저장 프로시듀어를 등록하는 매개변수 스타일은 저장 프로시듀어가 클라이언트 응용프로그램으로부터 데이터를 받는 방식을 결정합니다.

저장 프로시듀어 레지스터

CREATE PROCEDURE문을 사용하려면 반드시 다음을 선언해야 합니다.

- 프로시듀어 이름

- 각 매개변수의 모드, 이름 및 SQL 데이터 유형
- EXTERNAL 이름 및 위치
- PARAMETER STYLE

CREATE PROCEDURE는 다음 사항도 반드시 선언해야 합니다.

- FENCED를 수행할 것인지 또는 NOT FENCEF를 수행할 것인지 선언합니다.
- 프로시듀어 내용에 포함된 SQL문의 유형

DB2 계열의 호환성을 위한 전체 구문 및 옵션을 포함하여, CREATE PROCEDURE문에 대한 자세한 내용은 *SQL 참조서*에 있습니다. CREATE PROCEDURE문의 일반 사용법은 다음과 같습니다.

프로시듀어 이름: 고유한 수의 매개변수를 수용하는 프로시듀어에 대해 동일한 이름을 사용하여 저장 프로시듀어를 오버로드할 수 있습니다. DB2가 데이터 유형을 구별하지 않으므로 매개변수 데이터 유형에 기초한 ㅠ를 오버로드할 수 없습니다.

예를 들어, 다음과 같은 CREATE PROCEDURE문을 발행하는 것은 각각 한 매개변수 및 두 매개변수를 허용하므로 작동됩니다.u

```
CREATE PROCEDURE OVERLOAD (IN VAR1 INTEGER) ...
CREATE PROCEDURE OVERLOAD (IN VAR1 INTEGER, IN VAR2 INTEGER) ...
```

그러나, DB2는 같은 이름을 가진 첫번째 저장 프로시듀어로서 같은 수의 매개변수를 가지고 있으므로, 다음 예에서 두 번째 저장 프로시듀어를 등록하지 못합니다.

```
CREATE PROCEDURE OVERLOADFAIL (IN VAR1 INTEGER) ...
CREATE PROCEDURE OVERLOADFAIL (IN VAR2 VARCHAR(15)) ...
```

매개변수 모드: *명시적 매개변수*는 CREATE PROCEDURE문의 매개변수 목록에서 명시적으로 선언된 매개변수입니다. *암시적 매개변수*는 DB2에서 자동으로 제공되는 매개변수입니다. 예를 들어, PARAMETER STYLE GENERAL WITH NULLS 저장 프로시듀어는 명시적 매개변수에 대해 널 표시기의 배열을 자동으로 제공합니다. 저장 프로시듀어를 작성할 때 저장 프로시듀어에 대해 명시적 및 암시적 매개변수를 둘다 고려해야 합니다. 클라이언트 응용프로그램을 작성할 때 저장 프로시듀어에 대해 명시적 매개변수만 처리해야 합니다. 모든 명시적 매개변

수를 이름과 SQL 데이터 유형을 가진 IN, OUT 또는 INOUT 매개변수로 선언해야 합니다. CREATE PROCEDURE문 예에 대해서는 229 페이지의 『변수 선언 및 CREATE PROCEDURE 예』를 참조하십시오.

IN 클라이언트 응용프로그램으로부터 저장 프로시듀어로 값을 전달하지만, 제어가 클라이언트 응용프로그램으로 리턴될 때 클라이언트 응용프로그램으로 값을 리턴하지 않습니다.

OUT 저장 프로시듀어가 종료될 때 클라이언트 응용프로그램으로 전달된 값을 저장합니다.

INOUT

클라이언트 응용프로그램으로부터 저장 프로시듀어로 값을 전달하고, 저장 프로시듀어가 종료될 때 클라이언트 응용프로그램으로 값을 리턴합니다.

위치: CREATE PROCEDURE문의 EXTERNAL절은 저장 프로시듀어가 포함된 라이브러리의 위치를 데이터베이스 관리 프로그램에게 알려 줍니다. 라이브러리에 절대 경로를 지정하지 않거나, Java 저장 프로시듀어에 jar 이름을 지정하지 않으면 데이터베이스 관리 프로그램은 *함수 디렉토리*를 검색합니다. *함수 디렉토리*는 다음과 같이 사용자 운영 체제에 대해 정의된 디렉토리입니다.

Unix 운영 체제

```
sqllib/function
```

OS/2 또는 Windows 32 비트 운영 체제

instance_name\function, 여기서 *instance_name*은 *DB2INSTPROF* 인스턴스 고유의 레지스트리 설정 값을 표시합니다. *DB2INSTPROF*가 설정되지 않으면 *instance_name*은 *%DB2PATH%* 환경 변수의 값을 나타냅니다. *%DB2PATH%* 환경 변수의 기본값은 DB2를 설치한 경로입니다.

DB2rk *instance_name*\function에서 DB2가 저장 프로시듀어를 찾지 않으면 *PATH* 및 *LIBPATH* 환경 변수에 의해 정의된 디렉토리를 검색합니다.

예를 들어, DB2가 *DB2INSTPROF* 레지스트리 설정을 설정하지 않은 C:\sqllib 디렉토리에 설치된 Windows 32-비트 운영 체제 서버의 함수 디렉토리는 다음과 같습니다.

```
C:\sqllib\function
```

주: 사용자 라이브러리에 저장 프로시듀어 이름과 다른 이름을 제공해야 합니다. DB2가 검색 경로에서 라이브러리 위치를 찾으면 DB2는 FENCED DB2DARI 프로시듀어로서 라이브러리와 같은 이름을 가진 저장 프로시듀어를 실행합니다.

LANGUAGE C 저장 프로시듀어의 경우 다음을 지정하십시오.

- 다음 형식을 취하는 라이브러리 이름
  - 함수 디렉토리에 있는 라이브러리
  - 라이브러리 이름이 포함된 절대 경로
- 라이브러리에 있는 저장 프로시듀어의 진입점. 진입점을 지정하지 않으면 데이터베이스 관리 프로그램은 기본 진입점을 사용합니다. AIX에서의 IBM XL C 컴파일러는 사용자에게 기본 진입점으로서 라이브러리에 내보내진 함수 이름을 지정할 수 있도록 합니다. 저장 프로시듀어 호출 또는 CREATE FUNCTION 문에 라이브러리 이름만 지정되어 있는 경우 이 함수가 호출됩니다. 기본 진입점을 지정하려면 링크 단계에서 `-e` 옵션을 사용하십시오. 예를 들면, `-e` *funcname*은 *funcname*을 기본 진입점으로 만듭니다. 기타 UNIX 플랫폼에서는 이러한 메카니즘이 존재하지 않으므로 DB2는 기본 진입점을 라이브러리 자체와 같은 이름으로 간주합니다.

예를 들어, UNIX 기반 시스템에서 `mymod!proc8`은 데이터베이스 관리 프로그램을 `sqllib/function/mymod` 라이브러리로 지정하고, 이 라이브러리 내에서 `proc8` 진입점을 사용합니다. Windows 32비트 및 OS/2 운영 체제에서 `mymod!proc8`은 데이터베이스 관리 프로그램이 함수 디렉토리로부터 `mymod.dll`을 로드하도록 지정하고, 동적 링크 라이브러리(DLL)에 있는 `proc8()` 프로시듀어를 호출합니다.

LANGUAGE JAVA 저장 프로시듀어의 경우 다음을 사용하십시오.

```
[<jar-file-name>:]<class-name>.<method-name>
```

다음 목록은 Java 저장 프로시듀어에 대해 EXTERNAL 키워드를 정의합니다.

jar-file-name

데이터베이스에 설치된 jar 파일에 저장 프로시듀어 메소드가 들어 있으면 이 값을 포함해야 합니다. 키워드는 jar 파일의 이름을 나타내고 콜론(`:`)으로 구분됩니다. jar 파일명을 지정하지 않으면 데이터베이스 관리 프로

그램은 함수 디렉토리에 있는 클래스를 검색합니다. jar 파일 설치에 대한 세부사항은 746 페이지의 『Java 저장 프로시듀어 및 UDF』의 내용을 참조하십시오.

class-name

저장 프로시듀어 메소드가 들어 있는 클래스 이름. 클래스가 패키지의 일부이면 완전한 패키지 이름을 접두사로 포함시켜야 합니다.

method-name

저장 프로시듀어 메소드 이름

예를 들어, `MyPackage.MyClass.myMethod`를 지정하면 데이터베이스 관리 프로그램은 `MyPackage` 패키지의 `MyClass` 클래스에서 `myMethod` 메소드를 사용합니다. DB2는 `MyPackage`가 콜론(`:`) 분리문자 대신 마침표(`.`) 분리문자를 사용하므로 jar 파일 대신 패키지를 참조합니다. DB2는 `MyPackage` 패키지에 대해 함수 디렉토리를 검색합니다.

함수 디렉토리에 대한 세부사항은 216 페이지의 『위치』의 내용을 참조하십시오.

LANGUAGE: C/C++의 경우, CREATE PROCEDURE문에 LANGUAGE C를 선언하십시오. Java 저장 프로시듀어의 경우 LANGUAGE JAVA를 선언하십시오. Windows 32-비트 운영 체제에서 OLE 저장 프로시듀어에 대해 LANGUAGE OLE를 선언하십시오. COBOL 저장 프로시듀어의 경우 LANGUAGE COBOL을 선언하십시오. Fortran 또는 REXX 저장 프로시듀어의 경우, DB2DARI 저장 프로시듀어로서 저장 프로시듀어를 작성해야 합니다. DB2DARI 저장 프로시듀어 작성에 대해서는 857 페이지의 『부록C. DB2DARI 및 DB2GENERAL 저장 프로시듀어와 UDF』의 내용을 참조하십시오.

LANGUAGE C

데이터베이스 관리 프로그램은 ANSI C 호출 및 연결 규약을 사용하여 저장 프로시듀어를 호출합니다. 대부분의 C/C++ 저장 프로시듀어에 대해서 이 옵션을 사용하십시오.

LANGUAGE JAVA

데이터베이스 관리 프로그램은 Java 클래스의 메소드로서 저장 프로시듀어를 호출합니다. Java 저장 프로시듀어에는 이 옵션을 사용하십시오.

LANGUAGE OLE

데이터베이스 관리 프로그램은 저장 프로시듀어를 OLE 함수로 호출합니다. Windows 32-비트 운영 체제에서 모든 OLE 저장 프로시듀어에 대해 이 옵션을 사용하십시오. CREATE PROCEDURE문을 실행하기 전에 REGSVR32 명령을 사용하여 OLE 저장 프로시듀어를 포함하는 DLL을 등록해야 합니다. OLE 저장 프로시듀어는 FENCED 모드에서 실행되어야 합니다. OLE 저장 프로시듀어 사용에 대한 자세한 내용은 *응용프로그램 빌드 안내서*을 참조하십시오.

LANGUAGE COBOL

데이터베이스 관리 프로그램은 COBOL 호출 및 링크 규약을 사용하는 저장 프로시듀어를 호출합니다. COBOL 저장 프로시듀어에 대해 이 옵션을 사용하십시오.

서브루틴으로 매개변수 전달: PROGRAM TYPE SUB의 C 저장 프로시듀어는 인수를 서브루틴으로 승인합니다. 수치 데이터 유형 매개변수를 포인터로 전달하십시오. 문자 데이터 유형을 해당 길이의 배열로 전달하십시오. 예를 들어, 다음 C 저장 프로시듀어 서명은 INTEGER, SMALLINT 그리고 CHAR(3) 유형의 매개변수를 승인합니다.

```
int storproc (sqlint32 *arg1, short *arg2, char arg[4])
```

Java 저장 프로시듀어는 인수를 서브루틴으로만 승인할 수 있습니다. IN 매개변수를 단순 인수로 전달하십시오. OUT와 INOUT 매개변수를 단일 요소의 배열로 전달하십시오. 예를 들어, 다음 Java 저장 프로시듀어 서명은 INTEGER, SMALLINT 유형의 OUT 매개변수 그리고 CHAR(3) 유형의 INOUT 매개변수를 승인합니다.

```
int storproc (int arg1, short arg2[], String arg[])
```

기본 함수로 매개변수 전달: C 프로그램에서 주 함수와 같은 인수를 허용하는 저장 프로시듀어를 작성하려면 CREATE PROCEDURE문에 PROGRAM TYPE MAIN을 지정하십시오. PROGRAM TYPE MAIN의 저장 프로시듀어가 다음 스펙에 일치하도록 작성해야 합니다.

- DB2는 매개변수 배열의 첫번째 요소의 값을 저장 프로시듀어 이름으로 설정합니다.
- 저장 프로시듀어는 다음 두 인수를 통해 매개변수를 허용합니다.

– 매개변수 카운터 변수. 예 - *argc*
– 매개변수가 포함된 배열. 예 - *argv[]*

- 저장 프로시듀어는 공유 라이브러리로서 구축되어야 합니다.

PROGRAM TYPE MAIN 저장 프로시듀어에서 DB2는 *argv* 배열의 첫번째 요소 값을 (*argv[0]*)를 저장 프로시듀어의 이름으로 설정합니다. *argv* 배열의 나머지 요소는 저장 프로시듀어의 CREATE PROCEDURE문에 선언된 매개변수에 해당합니다. 예를 들어, 다음 embedded C 저장 프로시듀어는 IN 매개변수 하나를 *argv[1]*로 전달하고 두 OUT 매개변수를 *argv[2]*와 *argv[3]*으로 리턴합니다.

PROGRAM TYPE MAIN의 CREATE PROCEDURE문 예는 다음과 같습니다.

```
CREATE PROCEDURE MAIN_EXAMPLE (IN job CHAR(8),
   OUT salary DOUBLE, OUT errorcode INTEGER)
   DYNAMIC RESULT SETS 0
   LANGUAGE C
   PARAMETER STYLE GENERAL
   NO DBINFO
   FENCED
   READS SQL DATA
   PROGRAM TYPE MAIN
   EXTERNAL NAME 'spserver!mainexample'
```

저장 프로시듀어의 다음 코드를 *argv[1]*의 값을 CHAR(8) 호스트 변수 *injob*으로 복사하고 DOUBLE 호스트 변수 *outsalary*의 값을 *argv[2]*로 복사하며 SQLCODE를 *argv[3]*으로 리턴합니다.

```
EXEC SQL BEGIN DECLARE SECTION;
  char injob[9];
  double outsalary;
EXEC SQL END DECLARE SECTION;

SQL_API_RC SQL_API_FN main_example (int argc, char **argv)
{
  EXEC SQL INCLUDE SQLCA;

  /* argv[0] contains the procedure name, so parameters start at argv[1] */
  strcpy (injob, (char *)argv[1]);

  EXEC SQL SELECT AVG(salary)
    INTO :outsalary
    FROM employee
    WHERE job = :injob;

  memcpy ((double *)argv[2], (double *)&outsalary, sizeof(double));

  memcpy ((sqlint32 *)argv[3], (sqlint32 *)&SQLCODE, sizeof(sqlint32));
```

```
return (0);
} /* end main_example function */
```

PARAMETER STYLE: 표9에서는 DB2 버전 7용 CREATE PROCEDURE 문에서 허용되는 PARAMETER STYLE(수평축) 및 LANGUAGE(수직축)의 조합에 대해 요약합니다.

표 9. CREATE PROCEDURE: PARAMETER STYLE 및 LANGUAGE의 유효 조합

	GENERAL, GENERAL WITH NULLS	JAVA	DB2SQL	DB2DARI	DB2GENERAL
LANGUAGE C	Y	N	Y	Y	N
LANGUAGE JAVA	N	Y	N	N	Y
LANGUAGE OLE	N	N	Y	N	N
LANGUAGE COBOL	Y	N	Y	N	N

GENERAL

저장 프로시듀어는 클라이언트 응용프로그램에서 CALL문으로부터 호스트 변수로서 매개변수를 받습니다. 저장 프로시듀어는 널(NULL) 표시기를 직접 클라이언트 응용프로그램에 전달하지 않습니다. LANGUAGE C 또는 LANGUAGE COBOL 옵션을 지정할 때에도 GENERAL만 사용할 수 있습니다.

OS/390용 DB2 Universal Database 호환성: GENERAL은 SIMPLE과 같습니다.

PARAMETER STYLE GENERAL 저장 프로시듀어는 PROGRAM TYPE절의 값으로 표시된 방식으로 매개변수를 승인합니다. 다음 예는 PROGRAM TYPE SUBROUTINE을 사용하여 두 매개변수를 승인하는 PARAMETER STYLE GENERAL 저장 프로시듀어를 보여줍니다.

```
SQL_API_RC SQL_API_FN one_result_set_to_client
   (double *insalary, sqlint32 *out_sqlerror)
{
  EXEC SQL INCLUDE SQLCA;
```

```
    EXEC SQL WHENEVER SQLERROR GOTO return_error;

    EXEC SQL BEGIN DECLARE SECTION;
      double l_insalary;
    EXEC SQL END DECLARE SECTION;

    l_insalary = *insalary;
    *out_sqlerror = 0;

    EXEC SQL DECLARE c3 CURSOR FOR
      SELECT name, job, CAST(salary AS INTEGER)
      FROM staff
      WHERE salary > :l_insalary
      ORDER BY salary;

    EXEC SQL OPEN c3;
    /* Leave cursor open to return result set */

    return (0);

    /* Copy SQLCODE to OUT parameter if SQL error occurs */
    return_error:
    {
      *out_sqlerror = SQLCODE;
      EXEC SQL WHENEVER SQLERROR CONTINUE;
      return (0);
    }

  } /* end one_result_set_to_client function */
```

GENERAL WITH NULLS

사용자가 선언하는 각 매개변수에 대해 DB2는 해당 INOUT 매개변수 널 표시기를 할당합니다. GENERAL과 마찬가지로, 매개변수는 호스트 변수로서 전달됩니다. LANGUAGE C 또는 LANGUAGE COBOL 옵션을 지정할 때에도 GENERAL WITH NULLS만 사용할 수 있습니다.

OS/390용 DB2 Universal Database 호환성: GENERAL WITH NULLS은 SIMPLE WITH NULLS과 같습니다.

PARAMETER STYLE GENERAL WITH NULLS 저장 프로시듀어는 PROGRAM TYPE절의 값으로 나타난 방식으로 매개변수를 승인하고 선언된 매개변수마다 요소가 하나 있는 널 표시기의 배열을 할당합니다. 다음 SQL은 PROGRAM TYPE SUB을 사용하여 INOUT 매개변수 하나

와 OUT 매개변수 두개를 전달하는 PARAMETER STYLE GENERAL WITH NULLS 저장 프로시듀어를 등록합니다.

```
CREATE PROCEDURE INOUT_PARAM (INOUT medianSalary DOUBLE,
   OUT errorCode INTEGER, OUT errorLabel CHAR(32))
   DYNAMIC RESULT SETS 0
   LANGUAGE C
   PARAMETER STYLE GENERAL WITH NULLS
   NO DBINFO
   FENCED
   MODIFIES SQL DATA
   PROGRAM TYPE SUB
   EXTERNAL NAME 'spserver!inout_param'
```

다음 C 코드는 GENERAL WITH NULLS 저장 프로시듀어에서 필요한 널 표시기를 선언하고 사용하는 방법을 보여줍니다.

```
SQL_API_RC SQL_API_FN inout_param (double *inoutMedian,
  sqlint32 *out_sqlerror, char buffer[33], sqlint16 nullinds[3])
{
  EXEC SQL INCLUDE SQLCA;

  EXEC SQL WHENEVER SQLERROR GOTO return_error;

  if (nullinds[0] < 0)
  {
    /* NULL value was received as input, so return NULL output */
    nullinds[0] = -1;
    nullinds[1] = -1;
    nullinds[2] = -1;
  }
  else
  {
    int counter = 0;
    *out_sqlerror = 0;
    medianSalary = *inoutMedian;

    strcpy(buffer, "DECLARE inout CURSOR");
    EXEC SQL DECLARE inout CURSOR FOR
      SELECT CAST(salary AS DOUBLE) FROM staff
      WHERE salary > :medianSalary
      ORDER BY salary;

    nullinds[1] = 0;
    nullinds[2] = 0;

    strcpy(buffer, "SELECT COUNT INTO numRecords");
    EXEC SQL SELECT COUNT(*) INTO :numRecords
      FROM staff
      WHERE salary > :medianSalary;

    if (numRecords != 0)
    /* At least one record was found */
    {
      strcpy(buffer, "OPEN inout");
```

```
        EXEC SQL OPEN inout USING :medianSalary;

        strcpy(buffer, "FETCH inout");
        while (counter < (numRecords / 2 + 1)) {
          EXEC SQL FETCH inout INTO :medianSalary;

          *inoutMedian = medianSalary;
          counter = counter + 1;
        }

        strcpy(buffer, "CLOSE inout");
        EXEC SQL CLOSE inout;
      }
      else /* No records were found */
      {
        /* Return 100 to indicate NOT FOUND error */
        *out_sqlerror = 100;
      }
    }

    return (0);

    /* Copy SQLCODE to OUT parameter if SQL error occurs */
    return_error:
    {
      *out_sqlerror = SQLCODE;
      EXEC SQL WHENEVER SQLERROR CONTINUE;
      return (0);
    }

  } /* end inout_param function */
```

JAVA 저장 프로시듀어는 *SQLJ 루틴* 스펙과 일치하는 규약을 전달하는 매개변수를 사용합니다. 저장 프로시듀어는 호스트 변수로서 IN 매개변수를 받고, 단일 항목 배열로서 OUT 및 INOUT 매개변수를 받습니다. LANGUAGE JAVA 옵션을 지정할 때에도 JAVA만 사용할 수 있습니다.

DB2SQL

DB2SQL 저장 프로시듀어의 C 함수 정의는 다음 내재적 매개변수를 CREATE PROCEDURE문에서 선언된 매개변수 정의에 추가해야 합니다.

```
sqlint16 nullinds[n],   1
char sqlst[6],          2
char qualname[28],      3
char specname[19],      4
char diagmsg[71],       5
```

DB2는 다음 인수를 저장 프로시듀어로 전달합니다.

1. DB2는 내재적 SMALLINT INOUT 매개변수 배열을 명시적 매개변수의 널 표시기로 할당합니다. 배열 크기는 *n*으로, 여기서 *n*은 명시적 매개변수의 수를 나타냅니다.
2. SQLSTATE 값의 암시적 CHAR(5) OUT 매개변수.
3. 완전한 저장 프로시듀어 이름의 암시적 CHAR(27) IN 매개변수.
4. 저장 프로시듀어의 고유 이름에 대한 암시적 CHAR(18) IN 매개변수.
5. SQL 진단 문자열에 대한 암시적 CHAR(70) OUT 매개변수.

LANGUAGE C 또는 LANGUAGE COBOL 옵션을 지정할 때에도 DB2SQL만 지정할 수 있습니다. 예를 들어, 다음 CREATE PROCEDURE문은 PARAMETER STYLE DB2SQL 저장 프로시듀어를 등록합니다.

```
CREATE PROCEDURE DB2SQL_EXAMPLE (IN job CHAR(8), OUT salary DOUBLE)
   DYNAMIC RESULT SETS 0
   LANGUAGE C
   PARAMETER STYLE DB2SQL
   NO DBINFO
   FENCED
   READS SQL DATA
   PROGRAM TYPE SUB
   EXTERNAL NAME 'spserver!db2sqlexample'
```

다음 규칙을 사용하여 저장 프로시듀어를 작성하십시오.

- PARAMETER STYLE DB2SQL 저장 프로시듀어는 명시적 매개변수마다 요소가 하나 있는 널 표시기의 배열을 전달합니다. IN 또는 INOUT 매개변수에 대한 널 표시기 요소의 음의 값은 클라이언트 응용프로그램이 그 매개변수에 대한 널 값으로 전달되었음을 나타냅니다. 출력 매개변수가 널이 아님을 나타내려면 OUT 또는 INOUT 매개변수의 널 표시기 요소 값을 0으로 설정하십시오. 출력 매개변수가 널임을 나타내려면 OUT 또는 INOUT 매개변수에 대한 널 표시기 값을 -1로 설정하십시오.
- 앞에서 설명된 대로 DB2SQL 매개변수에 대해 저장 프로시듀어 서명에 인수를 추가하십시오.
- SQLCA에서 사용자 정의된 값을 클라이언트에게 리턴하도록 DB2SQL SQLSTATE(CHAR-5)와 진단 메시지(null-terminated CHAR(70)) 매개변수의 값을 설정할 수 있습니다.

예를 들어, 다음 embedded C 저장 프로시듀어는 PARAMETER STYLE DB2SQL 저장 프로시듀어의 코딩 스타일을 보여줍니다.

```
SQL_API_RC SQL_API_FN db2sql_example (
  char injob[9],          /* Input - CHAR(8)    */
  double *salary,         /* Output - DOUBLE    */
  sqlint16 nullinds[2],
  char sqlst[6],
  char qualname[28],
  char specname[19],
  char diagmsg[71]
)
{
  EXEC SQL INCLUDE SQLCA;

  if (nullinds[0] < 0)
  {
    /* NULL value was received as input, so return NULL output */
    nullinds[1] = -1;
    /* Set custom SQLSTATE to return to client. */
    strcpy(sqlst, "38100");
    /* Set custom message to return to client. */
    strcpy(diagmsg, "Received null input on call to DB2SQL_EXAMPLE.");

  }
  else
  {
    EXEC SQL SELECT (CAST(AVG(salary) AS DOUBLE))
      INTO :outsalary INDICATOR :outsalaryind
      FROM employee
      WHERE job = :injob;

    *salary = outsalary;
    nullinds[1] = outsalaryind;
  }
  return (0);
} /* end db2sql_example function */
```

다음 embedded C 클라이언트 응용프로그램은 DB2SQL_EXAMPLE 저장 프로시듀어를 호출하는 CALL문을 실행하는 방법을 보여줍니다. 예에는 CALL문의 각 매개변수에 대한 널 표시기가 들어 있습니다. 예는 널 표시기 *in_jobind*을 0으로 설정하여 널이 아닌 값이 호스트 변수 *in_job*으로 표현된 IN 매개변수의 저장 프로시듀어로 전달됨을 나타냅니다. OUT 매개변수의 널 표시기는 이런 매개변수의 저장 프로시듀어에 전달된 입력이 없음을 나타내도록 -1로 설정됩니다.

```
int db2sqlparm(char out_lang[9], char job_name[9])
{
   int testlang;

   EXEC SQL BEGIN DECLARE SECTION;
     /* Declare host variables for passing data to DB2SQL_EXAMPLE */
     char in_job[9];
     sqlint16 in_jobind;
     double out_salary = 0;
     sqlint16 out_salaryind;
```

```
EXEC SQL END DECLARE SECTION;

/*****************************************************\
* Call DB2SQL_EXAMPLE stored procedure               *
\*****************************************************/

testlang = strncmp(out_lang, "C", 1);
if (testlang != 0) {
   /* Only LANGUAGE C procedures can be PARAMETER STYLE DB2SQL,
      so do not call the DB2SQL_EXAMPLE stored procedure */
   printf("\nStored procedures are not implemented in C.\n"
          "Skipping the call to DB2SQL_EXAMPLE.\n");
}
else {
   strcpy(procname, "DB2SQL_EXAMPLE");
   printf("\nCALL stored procedure named %s\n", procname);

   /* out_salary is an OUT parameter, so set the
      null indicator to -1 to indicate no input value */
   out_salaryind = -1;

   strcpy(in_job, job_name);

   /* in_job is an IN parameter, so check to
       see if there is any input value */
   if (strlen(in_job) == 0)
   {
      /* in_job is null, so set the null indicator
         to -1 to indicate there is no input value */
      in_jobind = -1;
      printf("with NULL input, to return a custom
              SQLSTATE and diagnostic message\n");
   }
   else
   {
      /* in_job is not null, so set the null indicator
         to 0 to indicate there is an input value */
      in_jobind = 0;
   }

   /* DB2SQL_EXAMPLE is PS DB2SQL, so pass
      a null indicator for each parameter */
   EXEC SQL CALL :procname (:in_job:in_jobind,
       :out_salary:out_salaryind);

   /* DB2SQL stored procedures can return a custom
      SQLSTATE and diagnostic message, so instead of
      using the EMB_SQL_CHECK macro to check the value
      of the returned SQLCODE, check the SQLCA structure for
      the value of the SQLSTATE and the diagnostic message */

   /* Check value of returned SQLSTATE */
   if (strncmp(sqlca.sqlstate, "00000", 5) == 0) {
     printf("Stored procedure returned successfully.\n");
     printf("Average salary for job %s = %9.2f\n",
       in_job, out_salary);
   }
   else {
      printf("Stored procedure failed with SQLSTATE %s.\n",
        sqlca.sqlstate);
```

```
            printf("Stored procedure returned the following
              diagnostic message:\n");
            printf("  \"%s\"\n", sqlca.sqlerrmc);
         }
      }

     return 0;
   }
```

DB2GENERAL

저장 프로시듀어는 DB2 Java 저장 프로시듀어에서만 지원되는 매개변수 전달 규약을 사용합니다. LANGUAGE JAVA 옵션을 지정할 때에도 DB2GENERAL만 사용할 수 있습니다.

이식성의 증가를 위해서는 PARAMETER STYLE JAVA 규약을 사용하여 Java 저장 프로시듀어를 작성해야 합니다. DB2GENERAL 매개변수 스타일 저장 프로시듀어를 작성하는 것에 대한 세부사항은 857 페이지의 『부록C. DB2DARI 및 DB2GENERAL 저장 프로시듀어와 UDF』의 내용을 참조하십시오.

DB2DARI

저장 프로시듀어는 C 언어 호출 및 링크 규약과 일치하는 매개변수 전달 규약을 사용합니다. 이 옵션은 DB2 Universal Database에 의해서만 지원되며, LANGUAGE C 옵션을 지정할 때에도 이 옵션만 사용할 수 있습니다.

DB2 계열에서의 이식성을 증가시키기 위해서는 GENERAL 또는 GENERAL WITH NULLS 매개변수 스타일을 사용하여 LANGUAGE C 저장 프로시듀어를 작성해야 합니다. DB2DARI 매개변수 스타일 저장 프로시듀어를 작성하려면 857 페이지의 『부록C. DB2DARI 및 DB2GENERAL 저장 프로시듀어와 UDF』의 내용을 참조하십시오.

DBINFO 구조 전달: PARAMETER TYPE이 GENERAL, GENERAL WITH NULLS 또는 DB2SQL인 LANGUAGE C 저장 프로시듀어의 경우 저장 프로시듀어가 추가 매개변수를 허용하도록 작성할 수 있는 옵션이 있습니다. CREATE PROCEDURE문에서 DBINFO를 지정하여 클라이언트 응용프로그램이 DB2 클라이언트에 대한 정보가 포함된 DBINFO 구조를 호출 매개변수와 함께 저장 프로시듀어로 전달하도록 지시할 수 있습니다. DBINFO 구조에는 다음과 같은 값들이 포함되어 있습니다.

데이터베이스 이름
: 클라이언트가 연결된 데이터베이스의 이름.

응용프로그램 권한 부여 ID
: 응용프로그램 런타임 권한 부여 ID.

코드 페이지
: 데이터베이스의 코드 페이지.

스키마 이름
: 저장 프로시듀어에 적용되지 않습니다.

테이블 이름
: 저장 프로시듀어에 적용되지 않습니다.

컬럼 이름
: 저장 프로시듀어에 적용되지 않습니다.

데이터베이스 버전 및 릴리스
: 저장 프로시듀어를 호출하는 데이터베이스 서버의 버전, 릴리스 및 수정 레벨.

플랫폼 데이터베이스 서버의 플랫폼.

테이블 함수 결과 컬럼 번호
: 저장 프로시듀어에 적용되지 않습니다.

DBINFO 구조에 대한 세부사항은 446 페이지의 『DBINFO 구조』의 내용을 참조하십시오.

변수 선언 및 CREATE PROCEDURE 예

다음 예에서는 SAMPLE 데이터베이스에서 가상 시나리오에서 사용하는 CREATE PROCEDURE문 및 저장 프로시듀어 소스 코드를 보여줍니다.

IN 및 OUT 매개변수 사용: *empno*(SQL type VARCHAR)가 주어졌을 때 SAMPLE 데이터베이스의 EMPLOYEE 테이블에서 *lastname*(SQL type CHAR)을 리턴하는 Java 저장 프로시듀어 `GET_LASTNAME`을 작성한다고 하십시오. `myJar`로서 설치된 JAR에 포함된 `StoredProcedure` Java 클래스의 `getname` 메소드로서 프로시듀어를 작성합니다. 마지막으로 C로 코드화된 클라이언트 응용프로그램에서 저장 프로시듀어를 호출합니다.

1. 저장 프로시듀어 소스 코드에 두 개의 호스트 변수를 선언합니다.

```
String empid;
String name;
...
#sql { SELECT lastname INTO :empid FROM employee WHERE empno=:empid }
```

2. 다음 CREATE PROCEDURE문으로 저장 프로시듀어를 등록합니다.

```
CREATE PROCEDURE GET_LASTNAME (IN EMPID CHAR(6), OUT NAME VARCHAR(15))
    EXTERNAL NAME 'myJar:StoredProcedure.getname'
    LANGUAGE JAVA PARAMETER STYLE JAVA FENCED
    READS SQL DATA
```

3. C로 작성된 클라이언트 응용프로그램으로부터 저장 프로시듀어를 호출합니다.

```
EXEC SQL BEGIN DECLARE SECTION;
   struct name { short int; char[15] }
   char[7] empid;
EXEC SQL END DECLARE SECTION;
   ...
EXEC SQL CALL GET_LASTNAME (:empid, :name);
```

INOUT 매개변수 사용: 다음 예에서 *deptnumb*(SQL type SMALLINT)이 주어졌을 때 SAMPLE 데이터베이스의 ORG 테이블에서 *manager*(SQL type SMALLINT)를 리턴하는 C 저장 프로시듀어 *GET_MANAGER*를 작성한다고 하십시오.

1. *deptnumb*과 *manager*는 둘다 SQL 데이터 유형 SMALLINT이므로 클라이언트 응용프로그램에서 값을 수신하고 값을 리턴하는 저장 프로시듀어에서 단일 변수 *onevar*를 선언할 수 있습니다.

```
EXEC SQL BEGIN DECLARE SECTION;
  short onevar = 0;
EXEC SQL END DECLARE SECTION;
```

2. 다음 CREATE PROCEDURE문으로 저장 프로시듀어를 등록합니다.

```
CREATE PROCEDURE GET_MANAGER (INOUT onevar SMALLINT)
   EXTERNAL NAME 'stplib!getman'
   LANGUAGE C PARAMETER STYLE GENERAL FENCED
   READS SQL DATA
```

3. Java로 작성된 클라이언트 응용프로그램으로부터 저장 프로시듀어를 호출합니다.

```
short onevar = 0;
  ...
#SQL { CALL GET_MANAGER (:INOUT onevar) };
```

저장 프로시듀어의 SQL문

저장 프로시듀어는 SQL문을 포함될 수 있습니다. CREATE PROCEDURE문을 발행할 때 저장 프로시듀어가 포함하고 있는 SQL문 유형을 지정해야 합니다. 저장 프로시듀어를 등록할 때 값을 지정하지 않으면 데이터베이스 관리 프로그램는 MODIFIES SQL DATA를 사용합니다. 저장 프로시듀어에 사용되는 SQL 유형을 제한하기 위해 다음 네 개의 옵션 중 하나를 사용할 수 있습니다.

NO SQL

저장 프로시듀어가 SQL문을 실행할 수 없음을 나타냅니다. 저장 프로시듀어에서 SQL문을 실행하려고 하면 명령문은 SQLSTATE 38001을 리턴합니다.

CONTAINS SQL

SQL 데이터를 읽거나 수정할 수 없는 SQL문이 저장 프로시듀어에서 실행될 수 있음을 나타냅니다. 저장 프로시듀어가 SQL 데이터를 읽거나 수정할 수 있는 SQL문을 실행하려고 하면 명령문은 SQLSTATE 38004를 리턴합니다. 저장 프로시듀어에서 지원되지 않는 명령문은 SQLSTATE 38003을 리턴합니다.

READS SQL DATA

SQl을 수정할 수 없는 일부 SQL문은 저장 프로시듀어로 실행될 수 있음을 나타냅니다. 저장 프로시듀어가 데이터를 수정하는 SQL문을 실행하려고 하면 명령문은 SQLSTATE 38002를 리턴합니다. 저장 프로시듀어에서 지원되지 않는 명령문은 SQLSTATE 38003을 리턴합니다.

MODIFIES SQL DATA

저장 프로시듀어는 저장 프로시듀어에서 지원되지 않는 명령문을 제외한 모든 SQL문을 실행할 수 있음을 나타냅니다. 저장 프로시듀어가 저장 프로시듀어에서 지원되지 않는 SQL문을 실행하려고 하면 명령문은 SQLSTATE 38003을 리턴합니다.

CREATE PROCEDURE문에 대한 세부사항은 *SQL 참조서*의 내용을 참조하십시오.

중첩 저장 프로시듀어

중첩 저장 프로시듀어는 다른 저장 프로시듀어를 호출하는 저장 프로시듀어입니다. 이 기술을 다음과 같은 제한사항에서 DB2 응용프로그램에서 사용할 수 있습니다.

- 저장 프로시듀어는 LANGUAGE C 또는 LANGUAGE SQL로 카탈로그화되어야 합니다.
- 호출하는 저장 프로시듀어는 동일한 LANGUAGE절을 사용하여 카탈로그화될 수 있는 저장 프로시듀어만 호출할 수 있습니다. 중첩 호출에 대해서만 LANGUAGE C와 LANGUAGE SQL은 동일한 언어로 간주됩니다. 예를 들어, LANGUAGE C 저장 프로시듀어는 SQL 프로시듀어를 호출할 수 있습니다.
- 호출하는 저장 프로시듀어는 더 높은 SQL 데이터 액세스 레벨로 카탈로그화된 저장 프로시듀어를 호출할 수 없습니다. 예를 들어, CONTAINS SQL 데이터 액세스로 카탈로그화된 저장 프로시듀어는 NO SQL 또는 CONTAIN SQL 데이터 액세스로 카탈로그화된 저장 프로시듀어를 호출할 수 있지만, READS SQL DATA 또는 MODIFIES SQL DATA로 카탈로그화된 저장 프로시듀어를 호출할 수 없습니다.
- 최대 16 레벨의 중첩 저장 프로시듀어 호출을 지원할 수 있습니다. 예를 들어, 저장 프로시듀어 PROC1이 PROC2를 호출하고 PROC2가 PROC3을 호출하는 시나리오는 중첩 저장 프로시듀어의 3 레벨을 나타냅니다.
- 중첩의 모든 레벨에서 호출하고 호출된 저장 프로시듀어는 NOT FENCED로 카탈로그화될 수 없습니다.

중첩 SQL 프로시듀어는 하나 이상의 결과 세트를 클라이언트 응용프로그램이나 호출하는 저장 프로시듀어로 리턴할 수 없습니다. SQL 프로시듀어에서 결과 세트를 클라이언트 응용프로그램으로 리턴하려면, WITH RETURN TO CLIENT절을 사용하여 DECLARE CURSOR문을 실행하십시오. SQL 프로시듀어에서서 호출자가 클라이언트 응용프로그램이거나 호출하는 저장 프로시듀어인 호출자로 결과 세트를 리턴하려면, WITH RETURN TO CALLER절을 사용하여 DECLARE CURSOR문을 실행하십시오.

C로 작성된 중첩 embedd SQL 저장 프로시듀어와 중첩된 CLI 저장 프로시듀어는 결과 세트를 클라이언트 응용프로그램이나 호출하는 저장 프로시듀어로 리턴할 수 없습니다. 저장 프로시듀어가 있을 때 중첩 embedd SQL 저장 프로시듀어 또

는 중첩 CLI 저장 프로시듀어가 커서를 열어 놓으면 DB2는 커서를 닫습니다. 저장 프로시듀어에서 결과 세트를 리턴하는 것에 대한 자세한 내용은 251 페이지의 『저장 프로시듀어로부터 결과 세트 리턴』을 참조하십시오.

제한사항

저장 프로시듀어를 작성할 때 다음과 같은 제한사항이 있습니다.

- 예를 들어, Java에서 `System.out.println()`를 호출하거나 C/C++에서 `printf()`를 호출하거나 COBOL에서 `display`를 호출할 때는 표준 I/O 스트림을 사용하지 마십시오. 저장 프로시듀어는 백그라운드에서 수행되므로 화면에 작성할 수 없습니다. 그러나, 파일에 기록할 수 없습니다.
- 저장 프로시듀어를 등록할 때 같이 사용하는 CREATE PROCEDURE문이 허용하는 SQL문만 포함합니다. NO SQL, READS SQL DATA, CONTAINS SQL 또는 MODIFIES SQL DATA절을 사용하여 저장 프로시듀어를 카탈로그화하는 것에 대한 내용은 231 페이지의 『저장 프로시듀어의 SQL문』을 참조하십시오.
- 다음 조건 중 하나 또는 둘다 참일 때 저장 프로시듀어에서 COMMIT문을 사용할 수 없습니다.
  - NO SQL절을 사용하여 저장 프로시듀어를 카탈로그화합니다.
  - 저장 프로시듀어는 다중 사이트 갱신을 수행하는 응용프로그램에서 호출됩니다.
- 다음을 포함하여 저장 프로시듀어에서 연결 관련 명령문이나 명령을 실행할 수 없습니다.
  - BACKUP
  - CONNECT
  - CONNECT TO
  - CONNECT RESET
  - CREATE DATABASE
  - DROP DATABASE
  - FORWARD RECOVERY
  - RESTORE

- UNIX 기반 시스템에서 NOT FENCED 저장 프로시듀어는 DB2 Agent Process의 사용자 ID에서 수행됩니다. FENCED 저장 프로시듀어는 sqllib/adm의 .fenced 파일의 소유자로 설정된 db2dari 실행 파일의 사용자 ID에서 실행합니다. 이 사용자 ID는 저장 프로시듀어에 사용 가능한 시스템 자원을 제어합니다. db2dari 실행파일에 대한 세부사항은 사용자 플랫폼에 대한 *빠른 시작* 책을 참조하십시오.
- 매개변수의 SQL 데이터 유형이 달라도 같은 수의 매개변수를 허용하는 저장 프로시듀어를 오버로드할 수 없습니다.
- 저장 프로시듀어에는 현재 프로세스를 종료시킬 수 있는 명령이 포함될 수 없습니다. 저장 프로시듀어는 항상 현재 프로세스를 종료시키지 않고 클라이언트로 제어를 리턴해야 합니다.

OLE 자동화 저장 프로시듀어 작성

OLE(오브젝트 링크 및 삽입) 자동화는 Microsoft Corporation의 OLE 2.0 구조의 일부입니다. DB2는 OLE 자동화 오브젝트 메소드를 외부 저장 프로시듀어로서 호출할 수 있습니다. OLE 자동화 개요에 대한 내용은 471 페이지의 『OLE 자동 UDF 작성』을 참조하십시오.

OLE 자동화 오브젝트를 코드화한 후 CREATE PROCEDURE문을 사용하여 오브젝트 메소드를 저장 프로시듀어로 등록해야 합니다. OLE 자동화 저장 프로시듀어를 등록하려면 LANGUAGE OLE절과 함께 CREATE PROCEDURE문을 실행해야 합니다. 외부 이름은 OLE 자동화 오브젝트와 !(느낌표)로 구분된 메소드 이름을 식별하는 OLE progID로 구성됩니다.

다음 CREATE PROCEDURE문은 OLE 자동화 오브젝트 “db2smpl.salary”의 “median” 메소드에 대해 “median”라고 하는 OLE 자동화 저장 프로시듀어를 등록합니다.

```
CREATE PROCEDURE median (INOUT sal DOUBLE)
   EXTERNAL NAME 'db2smpl.salary!median'
   LANGUAGE OLE
   FENCED
   PARAMETER STYLE DB2SQL
```

OLE 메소드 구현에 대한 호출 규약은 C나 C++로 작성된 프로시듀어에 대한 규약과 동일합니다.

DB2는 SQL 유형과 OLE 자동화 유형간의 자동 유형변환을 처리합니다. 지원되는 OLE 자동화 유형과 SQL 유형간의 DB2 맵핑 목록에 대해 474 페이지의 표16을 참조하십시오. SQL 유형과 BASIC 또는 C/C++과 같은 OLE 프로그래밍 언어의 데이터 유형간의 DB2 맵핑 목록에 대해 475 페이지의 표17을 참조하십시오.

DB2와 OLE 자동화 저장 프로시듀어간에 전달된 데이터는 참조에 의한 호출로 전달됩니다. DB2는 이전에 참조된 테이블에 표시되어 있지 않은 DECIMAL 또는 LOCATORS와 같은 SQL 또는 부울이나 CURRENCY와 같은 OLE 자동화 유형을 지원하지 않습니다. BSTR에 맵핑된 문자 또는 그래픽 데이터는 데이터베이스 코드 페이지에서 UCS-2(유니코드라고도 하는 IBM 코드 페이지 13488) 스킴으로 맵핑됩니다. 리턴시 데이터는 다시 데이터베이스 코드 페이지로 변환됩니다. 이러한 변환은 데이터베이스 코드 페이지에 상관없이 발생합니다. 데이터베이스 코드 페이지에서 UCS-2로 또는 UCS-2에서 데이터베이스 코드 페이지로 변환할 코드 페이지 변환 테이블이 설치되어 있지 않은 경우, SQLCODE -332(SQLSTATE 57017)를 받게 됩니다.

예제 OUT 매개변수 저장 프로시듀어

다음은 OUT 호스트 변수 사용의 샘플 프로그램입니다. 클라이언트 응용프로그램은 SAMPLE 데이터베이스에 있는 사원의 중간 급여를 결정하는 저장 프로시듀어를 호출합니다. (중간의 정의는 해당 값의 반이 그 아래위에 있다는 것을 의미합니다.) 그러면 중간 급여가 OUT 호스트 변수를 사용하여 클라이언트 응용프로그램으로 다시 전달됩니다.

이 샘플 프로그램은 SAMPLE 데이터베이스에 있는 모든 사원의 중간 급여를 계산합니다. 중간 급여를 계산하기 위한 SQL 컬럼 함수가 없으므로, 중간 급여는 다음 알고리즘에 의해 대화식으로 처리될 수 있습니다.

1. 테이블에 있는 레코드 수(*n*)를 결정합니다.
2. 급여에 기초한 레코드를 순서화합니다.
3. $n / 2 + 1$ 행 위치의 레코드를 찾을 때까지 레코드를 페치합니다.
4. 이 레코드로부터 중간 급여를 읽습니다.

저장 프로시듀어 기술과 블로킹 커서를 사용하지 않는 응용프로그램은 그림5에서와 같이 네트워크를 통해 각 급여를 FETCH해야 합니다.

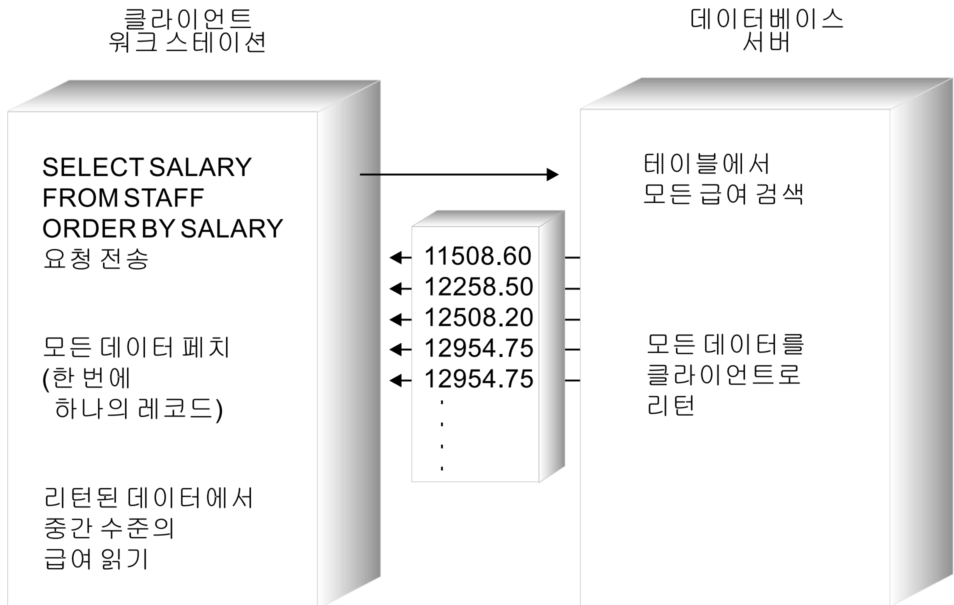


그림 5. 저장 프로시듀어 없는 중간 샘플

n ⁄ 2 + 1 행의 급여만 필요하므로, 응용프로그램은 모든 추가 데이터를 버리지만 *네트워크를 통해 전송된후에만 가능합니다.*

중간 급여만 클라이언트 응용프로그램으로 리턴하여, 저장 프로시듀어가 불필요한 데이터를 처리 및 제거하는 저장 프로시듀어 기술을 사용하여 응용프로그램을 설계할 수 있습니다. 그림6에 이러한 기능이 나와 있습니다.

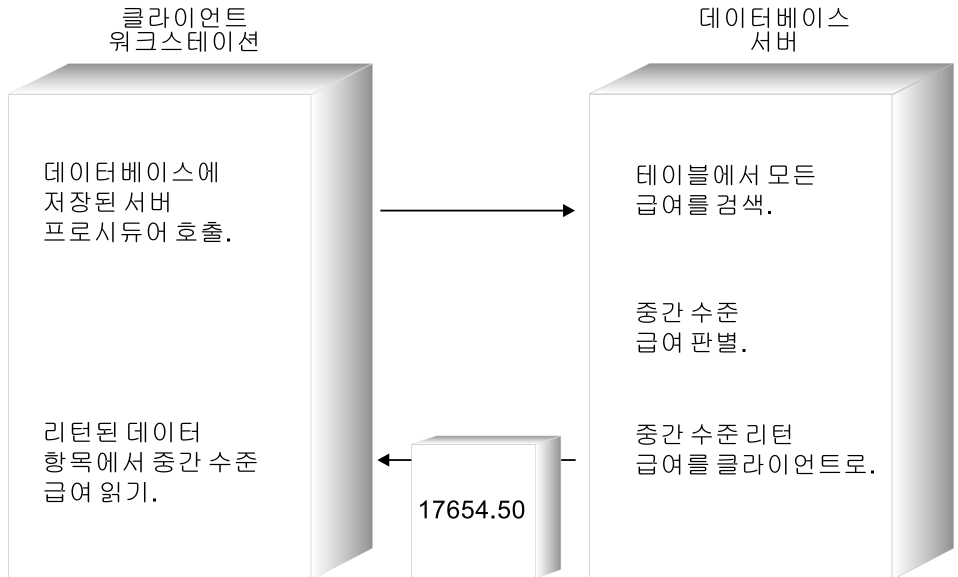


그림 6. 저장 프로시듀어를 사용하는 OUT 매개변수 샘플

『OUT 클라이언트 설명』에는 샘플 OUT 호스트 변수 클라이언트 응용프로그램 및 저장 프로시듀어가 표시됩니다. 샘플 프로그램은 다음과 같이 Java로 사용 가능합니다.

클라이언트 응용프로그램 Outcli.java

저장 프로시듀어 Outsrv.sqlj

샘플 프로그램은 다음과 같이 C로 사용 가능합니다.

클라이언트 응용프로그램 spclient.sqc

저장 프로시듀어 spserver.sqc

OUT 클라이언트 설명

1. 포함 파일. C 클라이언트 응용프로그램은 다음 파일들을 포함하고 있습니다.

   SQL SQL_TYP_FLOAT 기호를 정의합니다.

   SQLDA 설명자 영역을 정의합니다.

   SQLCA 오류 처리를 위해 통신 영역을 정의합니다.

   JDBC 클라이언트 응용프로그램은 다음 패키지를 가져옵니다.

java.sql.* 클라이언트의 JDBC 구현에서 나온 JDBC 클래스

java.math.BigDecimal
DB2 DECIMAL 데이터 유형에 대한 Java 지원을 제공합니다.

2. **데이터베이스에 연결**. 응용프로그램은 저장 프로시듀어를 호출하기 전에 데이터베이스에 연결되어야 합니다.
3. **자동 확약 끄기** 클라이언트 응용프로그램은 저장 프로시듀어를 호출하기 전에 자동 확약을 명시적으로 사용할 수 없게 만듭니다. 자동 확약을 사용할 수 없게 만들면 클라이언트 응용프로그램은 저장 프로시듀어에서 수행된 작업이 구간 복원되거나 확약되는지 제어할 수 있습니다. 이 예제의 저장 프로시듀어는 클라이언트 응용프로그램이 조건문을 사용하여 저장 프로시듀어에서 수행된 작업을 쉽게 확약하거나 구간 복원할 수 있도록 SQLCODE가 들어 있는 OUT 매개변수를 리턴합니다.
4. **호스트 변수 선언 및 초기화** 이 단계는 호스트 변수를 선언 및 초기화합니다. Java 프로그램은 저장 프로시듀어를 호출하기 전에 각 INOUT 또는 OUT 매개변수의 데이터 유형을 등록하고 모든 매개변수의 값을 초기화해야 합니다.
5. **저장 프로시듀어 호출**. 클라이언트 응용프로그램은 3개의 매개변수가 있는 CALL문을 사용하여 SAMPLE 데이터베이스의 OUTPARAM 저장 프로시듀어를 호출합니다.
6. **출력 매개변수 검색**. JDBC 클라이언트 응용프로그램은 저장 프로시듀어에 의해 리턴된 출력 매개변수의 값을 명시적으로 검색해야 합니다. C/C++ 클라이언트 응용프로그램에 대해 DB2는 클라이언트 응용프로그램에서 CALL문을 실행할 때 CALL문에서 사용된 호스트 변수의 값을 갱신합니다.
7. **리턴된 SQLCODE의 값을 확인하십시오**. 클라이언트 응용프로그램은 SQLCODE가 들어 있는 OUT 매개변수의 값을 확인하여 트랜잭션을 구간 복원할 것인지 아니면 확약할 것인지를 결정합니다.
8. **데이터베이스에서 연결해제**. DB2에서 각 연결에 할당된 시스템 자원을 해제할 수 있게 하려면 클라이언트 응용프로그램을 종료하기 전에 데이터베이스에 대한 연결을 명시적으로 닫아야 합니다.

CHECKERR 매크로/함수는 오류 체크 유틸리티입니다. 이 오류 체크 유틸리티의 위치는 사용된 프로그래밍 언어에 따라 달라질 수 있습니다.

C DB2 API를 호출하는 C 프로그램에 대해서는 utilapi.c에서 sqlInfoPrint 함수는 utilapi.h에서 API_SQL_CHECK로서 참조됩니다. C embedded SQL 프로그램에 대해서는 utilemb.sqc에서 sqlInfoPrint 함수는 utilemb.h에서 EMB_SQL_CHECK로서 참조됩니다.

Java SQL 오류는 SQLException으로서 드로우되고, 응용프로그램의 catch 블록에서 처리됩니다.

COBOL CHECKERR은 checkerr.cbl라고 하는 외부 프로그램입니다.

이 오류 체크 유틸리티에 대한 소스 코드에 대해서는 137 페이지의 『예제 프로그램에서 GET ERROR MESSAGE 사용』에서 자세한 내용을 참조하십시오.

예제 OUT 클라이언트 응용프로그램: Java

```
import java.sql.*;            // JDBC classes                          1
import java.math.BigDecimal;  // BigDecimal support for packed decimal type
class Spclient
{
  static String sql = "";
  static String procName = "";
  static String inLanguage = "";
  static CallableStatement callStmt;
  static int outErrorCode = 0;
  static String outErrorLabel = "";
  static double outMedian = 0;

  static
  {
    try
    {
      System.out.println();
      System.out.println("Java Stored Procedure Sample");
      Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
    }
    catch (Exception e)
    {
      System.out.println("\nError loading DB2 Driver...\n");
      e.printStackTrace();
    }
  }

  public static void main(String argv[])
  {
    Connection con = null;
    // URL is jdbc:db2:dbname
    String url = "jdbc:db2:sample";

    try
    {
      // connect to sample database
      // connect with default id/password
      con = DriverManager.getConnection(url);     2

      // turn off autocommit
      con.setAutoCommit(false);     3

      outLanguage(con);
      outParameter(con);
      inParameters(con);
      inoutParam(con, outMedian);
      resultSet(con);
      twoResultSets(con);
      allDataTypes(con);

      // rollback any changes to the database
      con.rollback();     8
      con.close();
    }
    catch (Exception e)
    {
      try { con.close(); } catch (Exception x) { }
      e.printStackTrace ();
    }
  } // end main

  public static void outParameter(Connection con)
    throws SQLException
  {
    // prepare the CALL statement for OUT_PARAM
```

```
      procName = "OUT_PARAM";
      sql = "CALL " + procName + "(?, ?, ?)";
      callStmt = con.prepareCall(sql);

      // register the output parameter       4
      callStmt.registerOutParameter (1, Types.DOUBLE);
0      callStmt.registerOutParameter (2, Types.INTEGER);
      callStmt.registerOutParameter (3, Types.CHAR);

0      // call the stored procedure       5
      System.out.println ("\nCall stored procedure named " + procName);
      callStmt.execute();

      // retrieve output parameters     6
      outMedian = callStmt.getDouble(1);
      outErrorCode = callStmt.getInt(2);
      outErrorLabel = callStmt.getString(3);

      if (outErrorCode == 0) {       7
        System.out.println(procName + " completed successfully");
        System.out.println ("Median salary returned from OUT_PARAM = "
          + outMedian);
      }
      else  { // stored procedure failed
        System.out.println(procName + " failed with SQLCODE "
          + outErrorCode);
        System.out.println(procName + " failed at " + outErrorLabel);
      }
    }
  }
```

예제 OUT 클라이언트 응용프로그램: C

```
#include <stdio.h>      1
#include <stdlib.h>
#include <sql.h>
#include <sqlda.h>
#include <sqlca.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
   /* Declare host variable for stored procedure name */
   char procname[254];

   /* Declare host variables for stored procedure error handling */
   sqlint32 out_sqlcode;      4
   char out_buffer[33];
EXEC SQL END DECLARE SECTION;

int main(int argc, char *argv[]) {

   EXEC SQL CONNECT TO sample;     2
   EMB_SQL_CHECK("CONNECT TO SAMPLE");

   outparameter();

   EXEC SQL ROLLBACK;
   EMB_SQL_CHECK("ROLLBACK");
   printf("\nStored procedure rolled back.\n\n");

   /* Disconnect from Remote Database */
   EXEC SQL CONNECT RESET;    8
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}

int outparameter() {
   /********************************************************\
   * Call OUT_PARAM stored procedure                        *
   \********************************************************/
   EXEC SQL BEGIN DECLARE SECTION;
     /* Declare host variables for passing data to OUT_PARAM */
     double out_median;
   EXEC SQL END DECLARE SECTION;

   strcpy(procname, "OUT_PARAM");
   printf("\nCALL stored procedure named %s\n", procname);

   /* OUT_PARAM is PS GENERAL, so do not pass a null indicator */
   EXEC SQL CALL :procname (:out_median, :out_sqlcode, :out_buffer);   5  6
   EMB_SQL_CHECK("CALL OUT_PARAM");
   /* Check that the stored procedure executed successfully */
   if (out_sqlcode == 0)     7
   {
     printf("Stored procedure returned successfully.\n");

     /**********************************************************\
      * Display the median salary returned as an output parameter *
```

```
    \*********************************************************/
      printf("Median salary returned from OUT_PARAM = %8.2f\n", out_median);

    }
    else
    { /* print the error message, roll back the transaction */
      printf("Stored procedure returned SQLCODE %d\n", out_sqlcode);
      printf("from procedure section labelled \"%s\".\n", out_buffer);
    }

    return 0;
}
```

OUT 저장 프로시듀어 설명

1. **신호 선언**. 프로시듀어는 3가지 매개변수를 리턴합니다. DOUBLE은 중간값을 위한 것이고 INTEGER는 SQLCODE를 위한 것이며 CHAR는 오류 메시지를 위한 것입니다. 각 언어의 프로그래밍 장에 지정된 DB2 유형 매핑을 사용하여 저장 프로시듀어 함수 정의에서 동등한 데이터 유형을 인수로 지정해야 합니다.
2. **급여별로 정렬된 CURSOR 선언하십시오**. 복수 데이터 행에 대해 작업하려면 C 저장 프로시듀어는 DECLARE CURSOR문을 실행하고 JDBC 저장 프로시듀어는 ResultSet 오브젝트를 작성합니다. ORDER BY SALARY절은 저장 프로시듀어를 통해 급여를 오름차순으로 검색합니다.
3. **전체 사원 수 결정**. 저장 프로시듀어는 COUNT 함수와 함께 간단한 SELECT 문을 사용하여 EMPLOYEE 테이블에 있는 사원 수를 검색합니다.
4. **중간 급여 FETCH**. 저장 프로시듀어는 중간 급여액을 변수에 지정할 때까지 연속적으로 FETCH문을 발행합니다.
5. **중간 급여를 출력 변수에 할당**. 중간 급여를 클라이언트 응용프로그램에 리턴하려면 값을 저장 프로시듀어 함수 또는 OUT 매개변수에 해당하는 메소드 선언에 할당합니다.
6. **클라이언트 응용프로그램으로 리턴**. PARAMETER STYLE DB2DARI 저장 프로시듀어만 클라이언트에 값을 리턴합니다. DB2DARI 저장 프로시듀어에 대해서는 857 페이지의『부록C. DB2DARI 및 DB2GENERAL 저장 프로시듀어와 UDF』에서 자세한 내용을 참조하십시오.

예제 OUT 매개변수 저장 프로시듀어: Java

```
import java.sql.*;              // JDBC classes
import COM.ibm.db2.jdbc.app.*;  // DB2 JDBC classes
import java.math.BigDecimal;  // Packed Decimal class

public class Spserver
{
   public static void outParameter (double[] medianSalary,
         int[] errorCode, String[] errorLabel) throws SQLException  1
   {   try
      {
         int numRecords;
         int counter = 0;
         errorCode[0] = 0; // SQLCODE = 0 unless SQLException occurs

         // Get caller's connection to the database
         Connection con = DriverManager.getConnection("jdbc:default:connection");
         errorLabel[0] = "GET CONNECTION";

         String query = "SELECT COUNT(*) FROM staff";
         errorLabel[0] = "PREPARE COUNT STATEMENT";
         PreparedStatement stmt = con.prepareStatement(query);
         errorLabel[0] = "GET COUNT RESULT SET";
         ResultSet rs = stmt.executeQuery();

         // move to first row of result set
         rs.next();

         // set value for the output parameter
         errorLabel[0] = "GET NUMBER OF RECORDS";
         numRecords = rs.getInt(1);   3

         // clean up first result set
         rs.close();
         stmt.close();

         // get salary result set
         query = "SELECT CAST(salary AS DOUBLE) FROM staff "
            + "ORDER BY salary";
         errorLabel[0] = "PREPARE SALARY STATEMENT";
         PreparedStatement stmt2 = con.prepareStatement(query);
         errorLabel[0] = "GET SALARY RESULT SET";
         ResultSet rs2 = stmt2.executeQuery();    2

         while (counter < (numRecords / 2 + 1))
         {
            errorLabel[0] = "MOVE TO NEXT ROW";
            rs2.next();   4
            counter++;
         }
         errorLabel[0] = "GET MEDIAN SALARY";
         medianSalary[0] = rs2.getDouble(1);    5

         // clean up resources
         rs2.close();
         stmt2.close();
         con.close();    6

      }
      catch (SQLException sqle)
      {
```

```
            errorCode[0] = sqle.getErrorCode();
        }
    }
}
```

예제 OUT 매개변수 저장 프로시듀어: C

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlca.h>
#include <sqludf.h>
#include <sql.h>
#include <memory.h>

/* Declare function prototypes for this stored procedure library */

SQL_API_RC SQL_API_FN out_param (double *, sqlint32 *, char *);  1

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
/* Declare host variables for basic error-handling */
  sqlint32 out_sqlcode;
  char buffer[33];

/* Declare host variables used by multiple stored procedures */
  sqlint16 numRecords;
  double medianSalary;
EXEC SQL END DECLARE SECTION;

SQL_API_RC SQL_API_FN out_param (double *outMedianSalary,
  sqlint32 *out_sqlerror, char buffer[33])
{
  EXEC SQL INCLUDE SQLCA;

  EXEC SQL WHENEVER SQLERROR GOTO return_error;

  int counter = 0;
  *out_sqlerror = 0;

  strcpy(buffer, "DECLARE c1");
  EXEC SQL DECLARE c1 CURSOR FOR    2
    SELECT CAST(salary AS DOUBLE) FROM staff
    ORDER BY salary;

  strcpy(buffer, "SELECT");
  EXEC SQL SELECT COUNT(*) INTO :numRecords FROM staff;   3
```

```
  strcpy(buffer, "OPEN");
  EXEC SQL OPEN c1;

  strcpy(buffer, "FETCH");
  while (counter < (numRecords / 2 + 1)) {
    EXEC SQL FETCH c1 INTO :medianSalary;   4

    /* Set value of OUT parameter to host variable */
    *outMedianSalary = medianSalary;   5
    counter = counter + 1;
  }

  strcpy(buffer, "CLOSE c1");
  EXEC SQL CLOSE c1;

  return (0);   6

  /* Copy SQLCODE to OUT parameter if SQL error occurs */
return_error:
  {
    *out_sqlerror = SQLCODE;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    return (0);
  }

} /* end out_param function */
```

코드 페이지 고려사항

코드 페이지 유의사항은 서버에 따라 다릅니다.

(코드 페이지 A와 같은) 클라이언트 프로그램이 (코드 페이지 Z와 같은) 다른 코드 페이지를 사용하는 데이터베이스에 액세스하는 원격 저장 프로시듀어를 호출할 때, 다음 이벤트가 발생합니다.

1. 입력 문자열 매개변수(클라이언트 응용프로그램에서 SQLDA로 정의되었는지 또는 호스트 변수로서 정의되었는지 나타내는 매개변수)는 응용프로그램 코드 페이지(A)에서 데이터베이스와 연관된 코드 페이지(Z)로 변환됩니다. FOR BIT DATA로서 SQLDA에 정의된 데이터에 대해 변환이 발생하지 않습니다.
2. 입력 매개변수가 변환되면 데이터베이스 관리 프로그램은 더 많은 코드 페이지 변환을 수행하지 않습니다.

   따라서, **데이터베이스와 같은 코드 페이지, 이 예에서는 코드 페이지 Z를 사용하여 저장 프로시듀어를 수행해야 합니다**. 데이터베이스와 같은 코드 페이지를 사용하여 서버 프로시듀어를 준비, 컴파일 및 바인드해 보는 것이 좋습니다.
3. 저장 프로시듀어가 완료될 때 데이터베이스 관리 프로그램은 출력 문자열 매개변수(호스트 변수로서 정의되었는지 또는 클라이언트 응용프로그램의 SQLDA에서 정의되었는지 나타내는 매개변수) 및 SQLCA 문자 필드를 데이터베이스 코드 페이지(Z)를 다시 응용프로그램 코드 페이지(A)로 변환합니다. FOR BIT DATA로서 SQLDA에 정의된 데이터에 대해 변환이 발생하지 않습니다.

주: 저장 프로시듀어의 매개변수가 서버에서 FOR BIT DATA로서 정의되어 있는 경우, SQLDA에서 명시적으로 지정되었는지에 관계없이 CALL문에 대해 OS/390용 DB2 Universal Database 또는 AS/400용 DB2 Universal Database 으로의 변환이 이루어지지 않습니다. (세부사항은 *SQL 참조서*에 있는 SQLDA 의절을 참조하십시오.)

이 주제에 대한 세부사항은 573 페이지의 『다른 코드 페이지 간의 변환』의 내용을 참조하십시오.

C++ 고려사항

C++에서 저장 프로시듀어를 작성할 때 다음 예에서와 같이 extern "C"를 사용하여 프로시듀어 이름을 선언할 수 있습니다.

```
extern "C" SQL_API_RC SQL_API_FN proc_name( short *parm1, char *parm2)
```

extern "C"는 C++ 컴파일러에 의해 함수 이름의 유형 데코레이션(또는 mangling)을 방지합니다. 이러한 데코레이션 없이 저장 프로시듀어를 호출할 때 함수 이름에 대해 모든 유형 데코레이션을 포함시켜야 합니다.

그래픽 호스트 변수 고려사항

매개변수 입력 또는 출력을 통해 그래픽 데이터를 받거나 리턴하는 C 또는 C++로 작성된 저장 프로시듀어는 일반적으로 WCHARTYPE NOCONVERT 옵션을 사용하여 사전 처리 컴파일됩니다. 이는 이러한 매개변수를 통해 전달된 그래픽 데이터가 wchar_t 프로세스 코드 형식이 아닌 DBCS 형식으로 간주되기 때문입니다. NOCONVERT를 사용한다는 것은, 저장 프로시듀어의 SQL문에서 조작되는 그래픽 데이터는 매개변수 데이터 형식과 일치하는 DBCS 형식이기도 합니다.

WCHARTYPE NOCONVERT를 이용하여 그래픽 호스트 변수와 데이터베이스 관리 프로그램 사이에 문자 코드 변환이 이루어집니다. 그래픽 호스트 변수의 데이터가 교체되지 않는 DBCS 문자로서 데이터베이스 관리 프로그램으로 전송 및 데이터베이스 관리 프로그램으로부터 수신됩니다. WCHARTYPE NOCONVERT를 사용하지 않을 경우 저장 프로시듀어의 wchar_t 형식으로 그래픽 데이터를 조작할 수 있습니다. 그러나, 수동으로 입출력 변환을 수행해야 합니다.

CONVERT는 FENCED 저장 프로시듀어에서 사용할 수 있으며, 저장 프로시듀어 내에서 SQL문에 있는 그래픽 데이터에 영향을 주지만 저장 프로시듀어의 인터페이스를 통한 데이터에는 영향을 주지 않습니다. NOT FENCED 저장 프로시듀어는 NOCONVERT 옵션을 사용하여 구축해야 합니다.

요약해서, 입력 또는 출력 매개변수를 통해 저장 프로시듀어로부터 리턴되거나 저장 프로시듀어로 전달되는 그래픽 데이터는, WCHARTYPE 옵션과 함께 사전 처리 컴파일된 방식에 관계없이, DBCS 형식이어야 합니다.

C 응용프로그램에서의 그래픽 데이터 처리에 대한 주요 정보는 694 페이지의 『C 및 C++에서의 그래픽 호스트 변수 처리』의 내용을 참조하십시오. EUC 코드 세트 및 응용프로그램 안내서에 대한 세부사항은 581 페이지의 『일본어 및 대만어 EUC 및 UCS-2 코드 세트 변환』의 내용을, 보다 자세한 내용은 586 페이지의 『저장 프로시듀어에 대한 고려사항』의 내용을 참조하십시오.

다중사이트 갱신 고려사항

CONNECT TYPE 2와 함께 응용프로그램이 호출하는 저장 프로시듀어는 동적으로 또는 정적으로 COMMIT 또는 ROLLBACK을 발행할 수 없습니다.

NOT FENCED 저장 프로시듀어

저장 프로시듀어는 CREATE PROCEDURE문에서 FENCED로서 저장 프로시듀어를 등록하는지 또는 NOT FENCED로서 등록하는지에 따라, *FENCED* 또는 *NOT FENCED* 저장 프로시듀어로서 수행될 수 있습니다.

NOT FENCED 저장 프로시듀어는 데이터베이스 관리 프로그램(DB2 Agent의 주소 공간)과 같은 주소 공간에서 수행됩니다. NOT FENCED로서 저장 프로시듀어를 수행하면, FENCED로서 수행할 때에 비해 성능이 향상됩니다. 왜냐하면 기본적으로 FENCED 저장 프로시듀어는 특수 DB2 프로세스에서 수행되기 때문입니다. 이 프로세스의 주소 공간은 DB2 System Controller과는 구별되기 때문입니다.

주:

1. NOT FENCED 저장 프로시듀어를 수행하면 성능이 향상될 수 있지만, 사용자 코드는 데이터베이스 제어 구조를 우연히 또는 고의로 손상시킬 수 있습니다. 성능상의 이점을 최대화해야 할 경우 NOT FENCED 저장 프로시듀어만 사용해야 합니다. NOT FENCED로 수행하기 전에 모든 저장 프로시듀어를 철저히 테스트하십시오.
2. NOT FENCED 저장 프로시듀어를 수행하는 동안 심각한 오류가 발생하면, 데이터베이스 관리 프로그램은 그 오류가 저장 프로시듀어 코드에서 발생한 것인지 아니면 데이터베이스 코드에서 발생한 것인지 알아본 후 적절한 복구 절차를 시도합니다.

디버깅을 하려면 *지역 FENCED 저장 프로시듀어*를 사용해 보십시오. 지역 FENCED 프로시듀어는 PARAMETER STYLE DB2DARI 프로시듀어입니다. 지역 FENCED 프로시듀어를 호출하려면, CALL <*library-name*>!<*entry-point*>를 실행하십시오. 여기서 *library-name*은 공유 라이브러리의 이름을 나타내고, *entry-point*는 저장 프로시듀어에 대한 공유 라이브러리의 입력 포인트를 나타냅니다. 공유 라이브러리의 이름과 입력 포인트가 같으면 CALL<*entry-point*>을 실행할 수 있습니다.

NOT FENCED 및 일반 FENCED 저장 프로시듀어는 추가 주소 공간에 대한 디버거 액세스를 제공함으로써 디버깅을 복잡하게 만듭니다. 지역 FENCED 저장 프로시듀어는 응용프로그램의 주소 공간에서 수행되므로, 디버거가 응용프로그램 코드와 저장 프로시듀어 코드 모두에 액세스할 수 있도록 합니다. 디버깅을 위해 지역 FENCED 저장 프로시듀어를 작동시키려면 다음 단계를 수행하십시오.

1. FENCED 저장 프로시듀어로서 이 저장 프로시듀어를 등록합니다.
2. `DB2_STPROC_ALLOW_LOCAL_FENCED` 레지스트리 변수를 `true`로 설정합니다. 레지스트리 변수에 대한 세부사항은 *관리 안내서: 구현*의 내용을 참조하십시오.
3. DB2 서버와 같은 머신에서 클라이언트 응용프로그램을 수행하십시오.

주: 지역 FENCED 저장 프로시듀어를 디버깅할 때에는 233 페이지의『제한사항』에 나열된 제한사항을 위반하는 명령문을 사용하지 않도록 하십시오. DB2는 지역 FENCED 저장 프로시듀어에 대한 호출을 클라이언트 응용프로그램의 서브루틴에 대한 호출로서 취급합니다. 따라서, 지역 FENCED 저장 프로시듀어에는 프로시듀어 내용에서 CONNECT문을 수행하는 것과 같이, 일반 저장 프로시듀어에 대한 제한사항을 위반하는 명령문이 포함될 수 있습니다.

NOT FENCED 저장 프로시듀어를 기록할 때 운영 체제에 따라 스레드 환경에서 수행할 수 있음을 기억해 두십시오. 따라서, 저장 프로시듀어는 완전히 re-entrant 이거나 이러한 변수로의 액세스가 직렬화되도록 정적 변수를 관리해야 합니다.

주: DB2는 저장 프로시듀어에 있는 정적 데이터가 후속 호출에서 재초기화되는지를 보증할 수 없으므로, 저장 프로시듀어에서 정적 데이터를 사용하지 말아야 합니다.

NOT FENCED 저장 프로시듀어는 WCHARTYPE NOCONVERT 옵션을 사용

하여 사전 처리 컴파일되어야 합니다. 696 페이지의『C 및 C++의 WCHARTYPE 사전 처리 컴파일러 옵션』에서 자세한 내용을 참조하십시오.

DB2는 NOT FENCED 저장 프로시듀어에서 다음 기능의 사용을 지원합니다.

- 16 비트
- 다중 스레딩
- 호출하거나 다른 저장 프로시듀어에서 호출되는 중첩 호출
- 클라이언트 응용프로그램 또는 호출자에게 결과 세트를 리턴하는 결과 세트
- REXX

다음 DB2 API 및 DB2 CLI API는 NOT FENCED 저장 프로시듀어에서 지원되지 않습니다.

- BIND
- EXPORT
- IMPORT
- PRECOMPILE PROGRAM
- ROLLFORWARD DATABASE

저장 프로시듀어로부터 결과 세트 리턴

하나 이상의 결과 세트를 DB2 CLI, ODBC, JDBC 또는 SQLJ 클라이언트 응용프로그램으로 리턴하기 위해 저장 프로시듀어를 코드화할 수 있습니다. 이러한 지원의 양상은 다음과 같습니다.

- DB2 CLI, ODBC, JDBC 및 SQLJ 클라이언트는 결과 세트를 허용할 수 있습니다.
- DataJoiner 버전 2 서버로부터 액세스할 수 있는 서버에 저장 프로시듀어가 상주해 있는 경우, Embedded SQL을 사용하는 DB2 클라이언트가 복수의 결과 세트를 허용할 수 있습니다. 호스트 및 AS/400 플랫폼의 저장 프로시듀어가 복수의 결과 세트를 DB2 Connect 클라이언트로 리턴할 수 있습니다. DB2 Universal Database 서버의 저장 프로시듀어는 복수의 결과 세트를 호스트 및 AS/400 클라이언트로 리턴할 수 있습니다. 자세한 내용은 DataJoiner나 호스트 또는 AS/400 플랫폼의 제품 문서를 참조하십시오.
- 클라이언트 응용프로그램은 리턴되는 결과 세트에 대해 기술할 수 있습니다.

- 결과 세트는 응용프로그램에 의해 직렬 방식으로 처리되어야 합니다. 커서는 첫 번째 결과 세트에서 자동으로 열리고, 특수 호출(DB2 CLI의 경우 SQLMoreResults, JDBC의 경우 getMoreResults, SQLJ의 경우 getNextResultSet)이 제공되어 결과 세트에서 커서를 닫고 다음 세트에서 커서를 엽니다.
- 저장 프로시듀어는 결과 세트에서 커서를 선언함으로써 결과 세트가 리턴됨을 나타내고, 결과 세트에서 커서를 열고 프로시듀어에서 나갈 때 커서를 연 상태로 둡니다. 한 개 이상의 커서가 열린 상태로 있으면 커서가 열린 순서대로 결과 세트가 리턴됩니다.
- 읽지 않거나 페치되지 않은 행만 다시 결과 세트로 전달됩니다.
- 결과 세트를 리턴하는 저장 프로시듀어는 FENCED 모드에서 수행되어야 합니다.
- COMMIT 또는 ROLLBACK은 WITH HOLD 커서를 제외한 모든 커서를 닫습니다.
- DB2CLI.PROCEDURES 테이블의 RESULT_SETS 컬럼은 저장 프로시듀어가 결과 세트를 나타냅니다. 저장 프로시듀어를 CREATE PROCEDURE문으로 선언하면 DYNAMIC RESULT SETS절은 저장 프로시듀어에서 리턴된 결과 세트의 수를 나타내도록 이 값을 설정합니다.

결과 세트 처리에 대한 세부사항은,

- DB2 CLI에서 *CLI Guide and Reference*을 참조하십시오.
- Java의 경우 DB2 Java 지원 웹 페이지인 http://www.ibm.com/software/data/db2/java/를 참조하여 JDBC 및 SQLJ 스펙으로 링크하십시오.

예제: 저장 프로시듀어로부터 결과 세트 리턴

이 샘플 저장 프로시듀어는 다음과 같은 지원되는 언어로 클라이언트 응용프로그램에 결과 세트를 리턴하는 방법을 보여줍니다.

C spserver.sqc

Java Spserver.java

이 샘플 저장 프로시듀어는 IN 매개변수를 하나 승인하고 OUT 매개변수 하나와 결과 세트 하나를 리턴합니다. 저장 프로시듀어는 IN 매개변수를 사용하여

SALARY가 IN 매개변수보다 큰 행의 STAFF 테이블에 대한 NAME, JOB 그리고 SALARY 컬럼 값이 들어 있는 결과 세트를 작성합니다.

1 CREATE PROCEDURE문의 DYNAMIC RESULT SETS절을 사용하여 저장 프로시듀어를 등록하십시오. 예를 들어, C의 embedded SQL로 작성된 저장 프로시듀어를 등록하려면 다음 명령문을 실행하십시오.

```
CREATE PROCEDURE RESULT_SET_CLIENT
   (IN salValue DOUBLE, OUT sqlCode INTEGER)
   DYNAMIC RESULT SETS 1
   LANGUAGE C
   PARAMETER STYLE GENERAL
   NO DBINFO
   FENCED
   READS SQL DATA
   PROGRAM TYPE SUB
   EXTERNAL NAME 'spserver!one_result_set_to_client'
```

2 C 저장 프로시듀어의 embedded SQL에 대해 DECLARE CURSOR와 OPEN CURSOR문을 사용하여 열린 커서를 작성하십시오. CLI 저장 프로시듀어에 대해 `SQLPrepare` 및 `SQLBindParameter` API를 사용하여 결과 세트를 작성하십시오. JDBC로 작성된 Java 저장 프로시듀어에 대해 `prepareStatement`와 `executeQuery` 메소드를 사용하여 결과 세트를 작성하십시오.

3 커서나 결과 세트를 닫지 않고 데이터베이스에 대한 연결을 닫으십시오. 이 단계는 C 저장 프로시듀어에서 embedded SQL에 적용되지 않습니다.

4 Java 저장 프로시듀어: PARAMETER STYLE JAVA 저장 프로시듀어 리턴이 있는 각 결과 세트에 대해 해당 *ResultSet[]* 인수를 저장 프로시듀어 메소드 서명에 포함시켜야 합니다.

C 예: SPSERVER.SQC(one_result_set_to_client)

```
SQL_API_RC SQL_API_FN one_result_set_to_client
  (double *insalary, sqlint32 *out_sqlerror)
{
  EXEC SQL INCLUDE SQLCA;

  EXEC SQL WHENEVER SQLERROR GOTO return_error;

  l_insalary = *insalary;
  *out_sqlerror = 0;

  EXEC SQL DECLARE c3 CURSOR FOR        2
    SELECT name, job, CAST(salary AS INTEGER)
    FROM staff
    WHERE salary > :l_insalary
    ORDER BY salary;

  EXEC SQL OPEN c3;                     2
  /* Leave cursor open to return result set */

  return (0);    3

  /* Copy SQLCODE to OUT parameter if SQL error occurs */
return_error:
  {
    *out_sqlerror = SQLCODE;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    return (0);
  }

} /* end one_result_set_to_client function */
```

Java 예: Spserver.java(resultSetToClient)

```
public static void resultSetToClient
     (double inSalaryThreshold, // double input
      int[] errorCode,            // SQLCODE output
      ResultSet[] rs)             // ResultSet output  4
      throws SQLException
{
   errorCode[0] = 0; // SQLCODE = 0 unless SQLException occurs

   try {
      // Get caller's connection to the database
      Connection con =
        DriverManager.getConnection("jdbc:default:connection");

      // get salary result set using a parameter marker
      String query = "SELECT name, job, CAST(salary AS DOUBLE) " +
              "FROM staff " +
              "WHERE salary > ? " +
              "ORDER BY salary";

      // prepare the SQL statement
      PreparedStatement stmt = con.prepareStatement(query);

      // set the value of the parameter marker (?)
      stmt.setDouble(1, inSalaryThreshold);

      // get the result set that will be returned to the client
      rs[0] = stmt.executeQuery();  2

      // to return a result set to the client, do not close ResultSet
      con.close();  3
   }

   catch (SQLException sqle)
   {
      errorCode[0] = sqle.getErrorCode();
   }
}
```

예제: 저장 프로시듀어로부터 결과 세트 허용

이 샘플 클라이언트 응용프로그램은 다음과 같이 지원되는 언어로 저장 프로시듀어에서 결과 세트를 승인하는 방법을 보여줍니다.

C(CLI 사용) spclient.c

Java Spclient.java

이 샘플 클라이언트 응용프로그램은 RESULT_SET_CLIENT 저장 프로시듀어를 호출하고 결과 세트를 하나 승인합니다. 그러면 클라이언트 응용프로그램은 결과 세트의 내용을 표시합니다.

1 CREATE PROCEDURE문에서 선언된 매개변수에 해당하는 인수가 있는 저장 프로시듀어를 호출합니다.

2 JDBC 응용프로그램은 `getNextResultSet` 메소드를 사용하여 저장 프로시듀어에서 첫번째 결과 세트를 승인합니다.

3 결과 세트에서 행을 패치하십시오. 샘플 CLI 클라이언트는 `while` 루프를 사용하여 결과 세트에서 모든 행을 페치하고 표시합니다. 샘플 JDBC 클라이언트는 결과 세트에서 모든 행을 패치하고 표시하는 `fetchAll`라는 클래스 메소드를 호출합니다.

CLI 예: SPCLIENT.C(one_result_set_to_client):

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlcli1.h>
#include <sqlca.h>
#include "utilcli.h"          /* Header file for CLI sample code */

     SQLCHAR      stmt[50];
     SQLINTEGER   out_sqlcode;
     char         out_buffer[33];
     SQLINTEGER   indicator;
     struct sqlca sqlca;
     SQLRETURN    rc,rc1  ;
     char         procname[254];
     SQLHANDLE    henv;  /* environment handle */
     SQLHANDLE    hdbc;  /* connection handle */
     SQLHANDLE    hstmt1; /* statement handle */
     SQLHANDLE    hstmt2; /* statement handle */
     SQLRETURN    sqlrc = SQL_SUCCESS;
     double       out_median;

     int oneresultset1(SQLHANDLE);

int main(int argc, char *argv[])
 {
     SQLHANDLE    hstmt; /* statement handle */
     SQLHANDLE    hstmt_oneresult; /* statement handle */

     char         dbAlias[SQL_MAX_DSN_LENGTH + 1] ;
     char         user[MAX_UID_LENGTH + 1] ;
     char         pswd[MAX_PWD_LENGTH + 1] ;

     /* Declare  variables for passing data to INOUT_PARAM */
     double inout_median;

    /* checks the command line arguments */
     rc = CmdLineArgsCheck1( argc, argv, dbAlias, user, pswd );
     if ( rc != 0 )  return( 1 ) ;

    /* allocate an environment handle */
    printf("\n    Allocate an environment handle.\n");
    sqlrc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv ) ;
    if ( sqlrc != SQL_SUCCESS )
    {   printf( "\n--ERROR while allocating the environment handle.\n" ) ;
        printf( "  sqlrc            = %d\n", sqlrc);
        printf( "  line             = %d\n", __LINE__);
        printf( "  file             = %s\n", __FILE__);
        return( 1 ) ;
    }

    /* allocate a database connection handle */
    printf("    Allocate a database connection handle.\n");
    sqlrc = SQLAllocHandle( SQL_HANDLE_DBC, henv, &hdbc ) ;

    HANDLE_CHECK( SQL_HANDLE_ENV, henv, sqlrc, &henv, &hdbc ) ;

    /* connect to the database */
    printf( "    Connecting to the database %s ...\n", dbAlias ) ;
    sqlrc = SQLConnect( hdbc,
                     (SQLCHAR *)dbAlias, SQL_NTS,
                     (SQLCHAR *)user, SQL_NTS,
                     (SQLCHAR *)pswd, SQL_NTS
```

```
               ) ;
  HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, sqlrc, &henv, &hdbc ) ;
  printf( "    Connected to the database %s.\n", dbAlias ) ;


  /* set AUTOCOMMIT off */
  sqlrc = SQLSetConnectAttr( hdbc,
                             SQL_ATTR_AUTOCOMMIT,
                             SQL_AUTOCOMMIT_OFF, SQL_NTS) ;
  HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, sqlrc, &henv, &hdbc ) ;

  /* allocate one or more statement handles */
  printf("    Allocate a statement handle.\n");
  sqlrc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt ) ;
  HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, sqlrc, &henv, &hdbc ) ;
  sqlrc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt_oneresult ) ;
  HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, sqlrc, &henv, &hdbc ) ;

 /*********************************************************\
 * Call oneresultsettocaller stored procedure              *
 \*********************************************************/
  rc = oneresultset1(hstmt_oneresult);
  rc = SQLFreeHandle( SQL_HANDLE_STMT, hstmt_oneresult ) ;
  HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;

  /* ROLLBACK, free resources, and exit */

   rc = SQLEndTran( SQL_HANDLE_DBC, hdbc, SQL_COMMIT ) ;
   HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;

   printf("\nStored procedure rolled back.\n\n");

  /* Disconnect from Remote Database */

   rc = SQLFreeHandle( SQL_HANDLE_STMT, hstmt ) ;
   HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;

   printf( "\n>Disconnecting .....\n" ) ;
   rc = SQLDisconnect( hdbc ) ;
   HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;

   rc = SQLFreeHandle( SQL_HANDLE_DBC, hdbc ) ;
   HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;

   rc = SQLFreeHandle( SQL_HANDLE_ENV,  henv ) ;
   if ( rc != SQL_SUCCESS ) return( SQL_ERROR )  ;

   return( SQL_SUCCESS ) ;
}

   int oneresultset1(hstmt)
   SQLHANDLE    hstmt; /* statement handle */
 {
 /*********************************************************\
 * Call one_result_set_to_client stored procedure          *
 \*********************************************************/

  double          insalary = 20000;
  SQLINTEGER      salary_int;
  SQLSMALLINT     num_cols;
  char            name[40];
  char            job[10];

  strcpy(procname, "RESULT_SET_CALLER");   1
```

```
    printf("\nCALL stored procedure:  %s\n", procname);

    strcpy((char*)stmt,"CALL RESULT_SET_CALLER ( ?,? )");
    rc = SQLPrepare(hstmt, stmt, SQL_NTS);
    STMT_HANDLE_CHECK( hstmt, rc);

    /* Bind the parameter to application variables () */
    rc = SQLBindParameter(hstmt, 1,
                          SQL_PARAM_INPUT, SQL_C_DOUBLE,
                          SQL_DOUBLE,0,
                          0, &insalary,
                          0, NULL);
    rc = SQLBindParameter(hstmt, 2,
                          SQL_PARAM_OUTPUT, SQL_C_LONG,
                          SQL_INTEGER,0,
                          0, &out_sqlcode,
                          0, NULL);

    STMT_HANDLE_CHECK( hstmt, rc);
    rc = SQLExecute(hstmt);
    rc1 = SQLGetSQLCA(henv, hdbc, hstmt, &sqlca);
    STMT_HANDLE_CHECK( hstmt, rc);

    rc = SQLNumResultCols( hstmt, &num_cols ) ;
    STMT_HANDLE_CHECK( hstmt, rc);
    printf("Result set returned %d columns\n", num_cols);

    /* bind columns to variables */
    rc = SQLBindCol( hstmt, 1, SQL_C_CHAR, name, 40, &indicator);
    STMT_HANDLE_CHECK( hstmt, rc);
    rc = SQLBindCol( hstmt, 2, SQL_C_CHAR, job, 10, &indicator);
    STMT_HANDLE_CHECK( hstmt, rc);
    rc = SQLBindCol( hstmt, 3, SQL_C_LONG, &salary_int, 0, &indicator);
    STMT_HANDLE_CHECK( hstmt, rc);

    /* fetch result set returned from stored procedure */
    rc = SQLFetch( hstmt );    2
    rc1 = SQLGetSQLCA(henv, hdbc, hstmt, &sqlca);

    STMT_HANDLE_CHECK( hstmt, rc);

    printf("\n--------Name---------,  --JOB--, ---Salary--\n");
    while (rc  == SQL_SUCCESS && rc != SQL_NO_DATA_FOUND )   3
    {
    printf("%20s,%10s,    %d\n",name,job,salary_int);

    rc = SQLFetch( hstmt );
    }

    STMT_HANDLE_CHECK( hstmt, rc);

   /* Check that the stored procedure executed successfully */
    if (rc == SQL_SUCCESS) {
     printf("Stored procedure returned successfully.\n");
     }
     else {
     printf("Stored procedure returned SQLCODE %d\n", out_sqlcode);
     }
    rc = SQLCloseCursor(hstmt);

    return(rc);
}
```

Java 예: Spclient.java(resultSetToClient):

```
// prepare the CALL statement for RESULT_SET_CLIENT
procName = "RESULT_SET_CLIENT";
sql = "CALL " + procName + "(?, ?)";  1
callStmt = con.prepareCall(sql);

// set input parameter to median value passed back by OUT_PARAM
callStmt.setDouble (1, outMedian);

// register the output parameter
callStmt.registerOutParameter (2, Types.INTEGER);

// call the stored procedure
System.out.println ("\nCall stored procedure named " + procName);
callStmt.execute();

// retrieve output parameter
outErrorCode = callStmt.getInt(2);

if (outErrorCode == 0) {
   System.out.println(procName + " completed successfully");
   ResultSet rs = callStmt.getResultSet();  2
   while (rs.next()) {
      fetchAll(rs);  3
   }

   // close ResultSet
   rs.close();
}
else  { // stored procedure failed
   System.out.println(procName + " failed with SQLCODE "
     + outErrorCode);
}
```

문제점 해결

저장 프로시듀어 응용프로그램이 제대로 실행되지 못하면 다음 사항을 확인하십시오.

- 저장 프로시듀어가 맞는 호출 순서, 컴파일 옵션 등을 사용하여 작성되었는지 확인합니다.
- 응용프로그램이 같은 워크스테이션에서 클라이언트 응용프로그램 및 저장 프로시듀어 모두에서 지역적으로 실행되는지 확인합니다.
- 저장 프로시듀어는 *응용프로그램 빌드 안내서*의 지시를 따라 적절한 위치에 저장됩니다.

  예를 들어, OS/2 환경에서 FENCED 저장 프로시듀어의 동적 링크 라이브러리는 데이터베이스 서버의 `instance_name\function` 디렉토리에 있습니다.

- DB2 CLI 및 JDBC로 작성된 경우를 제외하고, 응용프로그램은 데이터베이스로 바인드됩니다.
- 저장 프로시듀어는 SQLCA 오류 클라이언트를 클라이언트 응용프로그램으로 정확히 리턴합니다.
- 저장 프로시듀어 함수 이름은 *대소문자를 구분*하며, 클라이언트 및 서버에서 정확히 일치해야 합니다.
- CREATE PROCEDURE문으로 저장 프로시듀어를 등록할 경우, 저장 프로시듀어 함수 이름은 라이브러리 이름과 일치해서는 안 됩니다.

  예를 들어, 데이터베이스 관리 프로그램은 DB2DARI 함수로서 Windows 32비트 운영 체제 라이브러리 `myfunc.dll`에 포함된 `myfunc` 저장 프로시듀어를 실행하며, 이 때 연관된 CREATE PROCEDURE문에 지정된 값은 무시됩니다.

주: Java 저장 프로시듀어 디버깅에 대해서는 743 페이지의 『Java에서 저장 프로시듀어 디버깅』의 내용을 참조하십시오.

다른 응용프로그램에서처럼, 컴파일러와 함께 제공된 디버거를 사용하여 지역 FENCED 저장 프로시듀어를 디버그할 수 있습니다. 제공된 디버거 사용에 대해서는 컴파일러 문서를 참조하십시오.

예를 들어, Windows NT상에서 Visual Studio™와 함께 제공된 디버거를 사용하려면 다음 단계를 수행하십시오.

단계 1. `DB2_STPROC_ALLOW_LOCAL_FENCED` 레지스트리 변수를 `true`로 설정합니다.

단계 2. `-Zi` 및 `-Od` 플래그를 이용하여 저장 프로시듀어 DLL에 대한 소스 파일을 컴파일한 후, `-DEBUG` 옵션을 사용하여 DLL을 링크하십시오.

단계 3. 결과 DLL을 서버의 *instance_name* `\function` 디렉토리로 복사하십시오.

단계 4. Visual Studio 디버거를 이용하여 서버상의 클라이언트 응용프로그램을 호출하십시오. 클라이언트 응용프로그램 `outcli.exe`의 경우, 다음 명령을 입력하십시오.

```
msdev spclient.exe
```

단계 5. Visual Studio 디버거 창이 열리면 **프로젝트 —> 설정값**을 선택하십시오.

단계 6. **디버그** 탭을 클릭하십시오.

단계 7. **카테고리** 화살표를 클릭하고 **추가 DLL**을 선택하십시오.

단계 8. **새로 작성** 버튼을 클릭하여 새로운 모듈을 작성하십시오.

단계 9. **열람** 버튼을 클릭하면 **열람** 창이 열립니다.

단계 10. `spserver.dll` 모듈을 선택하고 **확인**을 클릭하면 **설정값** 창이 열립니다.

단계 11. 저장 프로시듀어에 대한 소스 파일을 열고 중지점을 설정하십시오.

단계 12. 가기 버튼을 클릭하십시오. Visual Studio 디버거는 저장 프로시듀어가 호출될 때 중지됩니다.

단계 13. 이 지점에서 사용자는 Visual Studio 디버거를 사용하여 저장 프로시듀어를 디버그할 수 있습니다.

Visual Studio 디버거 사용에 대해서는 Visual Studio 제품 문서를 참조하십시오.

제8장 SQL 프로시듀어 작성

SQL 프로시듀어 및 외부 프로시듀어 비교 264
유효한 SQL 프로시듀어 본문 명령문 . . . 265
CREATE PROCEDURE문 실행 267
SQL 프로시듀어에서 조건 조절 268
조건 핸들러 선언 268
SIGNAL 및 RESIGNAL문 271
SQL 프로시듀어의 SQLCODE 및 SQLSTATE 변수 271
SQL 프로시듀어에서 동적 SQL 사용 . . . 272
중첩 SQL 프로시듀어 274
중첩 SQL 프로시듀어간의 매개변수 전달 274
중첩 SQL 프로시듀어에서 결과 새트 리턴 274
중첩 SQL 프로시듀어에 제한사항 . . . 275
SQL 프로시듀어로부터 결과 세트 리턴 . . 276
클라이언트 및 호출자 결과 세트 리턴 276
클라이언트 결과 세트 리턴. 277
호출자 결과 세트 리턴 277
호출자 결과 세트 리턴 277
SQL 프로시듀어 디버깅 279
SQL 프로시듀어 오류 메시지 표시 . . . 279
중간 파일을 사용한 SQL 프로시듀어 디버깅 282
SQL 프로시듀어의 예 283

*SQL 프로시듀어*는 CREATE PROCEDURE문에 프로시듀어의 논리가 들어 있는 저장 프로시듀어입니다. 코드가 들어 있는 CREATE PROCEDURE문의 부분을 *프로시듀어 본문*이라고 합니다.

SQL 프로시듀어를 작성하려면 다른 DDL문과 같은 CREATE PROCEDURE문을 실행하기만 하면 됩니다. IBM DB2 Stored Procedure Builder를 사용하여 DB2에 저장 프로시듀어를 정의하고, SQL 프로시듀어에 대해 소스 명령문을 지정하면 실행할 프로시듀어를 준비할 수 있습니다. IBM DB2 Stored Procedure Builder에 대한 세부사항은 289 페이지의 『제9장 IBM DB2 Stored Procedure Builder』의 내용을 참조하십시오.

이 장에서는 프로시듀어 본문이 들어 있는 CREATE PROCEDURE문을 작성하는 방법을 설명합니다. CREATE PROCEDURE문 및 프로시듀어 본문의 구문에 대한 자세한 내용은 *SQL 참조서*를 참조하십시오. IBM DB2 Stored Procedure Builder를 사용하여 SQL 프로시듀어를 작성하는 방법에 대한 자세한 내용은 289 페이지의 『제9장 IBM DB2 Stored Procedure Builder』를 참조하십시오.

© Copyright IBM Corp. 1993, 2000

SQL 프로시듀어 및 외부 프로시듀어 비교

외부 저장 프로시듀어 정의처럼 SQL 프로시듀어 정의는 다음과 같은 정보를 제공합니다.

- 프로시듀어명
- 매개변수 속성
- 프로시듀어를 작성하는 언어. SQL 프로시듀어에 대해 언어는 SQL입니다.
- 프로시듀어의 특정 이름과 프로시듀어에서 리턴된 결과 세트와 같은 프로시듀어에 대한 기타 정보.

외부 저장 프로시듀어의 CREATE PROCEDURE문과는 달리 SQL 프로시듀어의 CREATE PROCEDURE문은 EXTERNAL절을 지정하지 않습니다. 대신 SQL 프로시듀어에는 저장 프로시듀어의 소스 명령문이 들어 있는 프로시듀어 본문이 있습니다.

다음 예는단순한 저장 프로시듀어의 CREATE PROCEDUR문을 보여줍니다. 프로시듀어 이름, 프로시듀어와 주고 받는 매개변수 목록 그리고 LANGUAGE 매개변수는 모든 저장 프로시듀어에서 공통입니다. 그러나, 프로시듀어 본문을 형성하는 SQL의 LANGUAGE 값과 BEGIN...END 블록은 SQL 프로시듀어에 고유합니다.

```
CREATE PROCEDURE UPDATE_SALARY_1                1
(IN EMPLOYEE_NUMBER CHAR(6),                    2
IN RATE INTEGER)                                2
LANGUAGE SQL                                    3
BEGIN
   UPDATE EMPLOYEE                              4
   SET SALARY = SALARY * (1.0 * RATE / 100.0 )
   WHERE EMPNO = EMPLOYEE_NUMBER;
END
```

이전 예에 대한 주:

1 저장 프로시듀어 이름은 UPDATE_SALARY_1입니다.

2 두 매개변수에는 CHAR(6)와 INTEGER의 데이터 유형이 있습니다. 둘다 입력 매개변수입니다.

3 LANGUAGE SQL은 이것이 SQL 프로시듀어임을 나타내므로 프로시듀어 본문은 다른 매개변수를 따릅니다.

4 프로시듀어 본문은 직원 테이블에서 행을 갱신하는 단일 SQL UPDATE문으로 구성됩니다.

SL 본문에서 OUT 매개변수를 표현식에 있는 값으로 사용할 수 없습니다. 오로지 지정문을 사용하여 값을 OUT 매개변수에 지정하거나 SELECT, VALUES 및 FETCH문의 INTO절에서 목표 변수로 지정할 수 있습니다. IN 매개변수를 지정 또는 INTO절의 목표로 사용할 수 없습니다.

유효한 SQL 프로시듀어 본문 명령문

프로시듀어 본문은 단일 SQL 프로시듀어 명령문으로 구성됩니다. 프로시듀어 본문에서 사용할 수 있는 명령문 유형에는 다음이 포함됩니다.

지정 명령문
: 값을 출력 매개변수에 지정하거나 프로시듀어 본문에서만 정의되고 사용되는 변수인 SQL 변수에 지정합니다. 값을 IN 매개변수에 지정할 수 없습니다.

CASE문
: 하나 이상의 *조건* 평가를 기반으로 실행 경로를 선택합니다. 이 명령문은 *SQL 참조서*에서 설명된 CASE 표현식과 비슷합니다.

FOR문
: 테이블의 각 행에 대해 명령문 또는 명령문 그룹을 실행합니다.

GET DIAGNOSTICS문
: GET DIAGNOSTICS문은 이전 SQL문에 대한 정보를 리턴합니다.

GOTO문
: 프로그램 제어를 SQL 루틴에서 사용자가 정의한 레이블로 전송합니다.

IF문
: 조건의 평가를 기반으로 실행 경로를 선택합니다.

ITERATE문
: 제어 흐름을 레이블이 붙은 블록이나 루프로 전달합니다.

LEAVE문
: 프로그램 제어를 루프나 코드 블록 밖으로 전송합니다.

LOOP문

명령문 또는 명령문 그룹을 여러번 실행합니다.

REPEAT문

명령문 또는 명령문 그룹을 검색 조건이 참이 될때까지 실행합니다.

RESIGNAL문

RESIGNAL문은 조건 핸들러에서 오류 또는 경고 조건을 다시 신호하기 위해 사용됩니다. 그러면 선택적 메시지 텍스트와 함께 오류 또는 경고를 지정된 SQLSTATE와 더불어 리턴됩니다.

RETURN문

SQL 프로시듀어에서 호출자로 제어를 리턴합니다. 정수 값을 호출자에게 리턴할 수도 있습니다.

SIGNAL문

SIGNAL문은 오류 또는 경고 조건을 신호하기 위해 사용됩니다. 그러면 선택적 메시지 텍스트와 함께 오류 또는 경고를 지정된 SQLSTATE와 더불어 리턴됩니다.

SQL문

SQL 프로시듀어 본문에는 829 페이지의 『부록A. 지원되는 SQL문』에 나열된 SQL문이 들어 있습니다.

WHILE문

지정된 조건이 참인 동안 명령문 또는 명령문 그룹을 반복 실행합니다.

복합 텍스트 명령문

이 목록에는 SQL 선언, 조건 핸들러 또는 커서 선언뿐만 아니라 다른 명령문 유형을 하나 이상 포함할 수 있습니다.

SQL 프로시듀어 본문에 허용되는 SQL문의 완전한 목록에 대해 829 페이지의 『부록A. 지원되는 SQL문』을 참조하십시오. 이런 명령문의 자세한 설명 및 그문에 대해 *SQL 참조서*를 참조하십시오.

CREATE PROCEDURE문 실행

CREATE PROCEDURE문을 DB2 명령행 프로세서 (DB2 CLP) 스크립트로 실행하려면 스크립트에서 SQL문에 대해 대체 종료 문자를 사용해야 합니다. DB2 CLP 스크립트의 기본 문자인 세미콜론(';') 문자는 SQL 프로시듀어 본문에서 SQL 문을 종료합니다.

DV2 CLP 스크립트에서 대체 종료 문자를 사용하려면 표준 SQL문에서 사용되지 않은 문자를 선택하십시오. 다음 예에서 기호('@')는 script.db2라고 하는 DB2 CLP 스크립트의 종료 문자로 사용되었습니다.

```
    CREATE PROCEDURE UPDATE_SALARY_IF
    (IN employee_number CHAR(6), IN rating SMALLINT)
    LANGUAGE SQL
    BEGIN
      DECLARE not_found CONDITION FOR SQLSTATE '02000';
      DECLARE EXIT HANDLER FOR not_found
         SIGNAL SQLSTATE '20000' SET MESSAGE_TEXT = 'Employee not found';

      IF (rating = 1)
        THEN UPDATE employee
          SET salary = salary * 1.10, bonus = 1000
          WHERE empno = employee_number;
      ELSEIF (rating = 2)
        THEN UPDATE employee
          SET salary = salary * 1.05, bonus = 500
          WHERE empno = employee_number;
      ELSE UPDATE employee
          SET salary = salary * 1.03, bonus = 0
          WHERE empno = employee_number;
      END IF;
    END
@
```

명령행에서 DB2 CLP 스크립트를 처리하려면 다음 구문을 사용하십시오.

db2 -td*term-char* -vf *script-name*

여기서 *term-char*는 종료하는 문자를 나타내고 *script-name*은 처리할 DB2 CLP 스크립트의 이름을 나타냅니다. 이전 스크립트를 처리하려면 CLP에서 다음 명령을 실행하십시오.

```
db2 -td@ -vf script.db2
```

SQL 프로시듀어에서 조건 조절

*조건 핸들러*는 *조건*이 발생할 때 SQL 프로시듀어의 동작을 결정합니다. 일반 DB2 조건, 특정 SQLSTATE 값에 대해 정의된 *조건* 또는 특정 SQLSTATE 값에 대해 SQL 프로시듀어에서 조건 핸들러를 하나 이상 선언할 수 있습니다. 일반 *조건* 및 자신의 *조건*을 정의하는 것에 대한 자세한 내용은『조건 핸들러 선언』을 참조하십시오.

SQL 프로시듀어에 있는 명령문이 SQLWARNING 또는 NOT FOUND *조건*을 실행하고 각각의 *조건*에 핸들러를 선언한 경우 DB2는 해당 핸들러로 제어를 전달합니다. 그 특정 *조건*에 대해 핸들러를 선언하지 않으면 DB2는 *조건*에 대해 SQLSTATE 및 SQLCODE 변수를 해당 값으로 설정하고 제어를 프로시듀어 본문에 있는 다음 명령문으로 전달합니다.

SQL 프로시듀어에 있는 명령문이 SQLEXCEPTION *조건*을 일으키고 특정 SQLSTATE 또는 SQLEXCEPTION *조건*에 대해 핸들러를 선언하면 DB2는 그 핸들러로 제어를 전달합니다. DB2가 핸들러를 성공적으로 실행하면 SQLSTATE 및 SQLCODE의 값은 각각 ‘00000’과 0을 리턴합니다.

SQL 프로시듀어에 있는 명령문이 SQLEXCEPTION *조건*을 일으키고 특정 SQLSTATE 또는 SQLEXCEPTION에 대해 핸들러를 선언하지 않으면 DB2는 SQL 프로시듀어를 종료하고 클라이언트로 리턴합니다.

조건 핸들러 선언

핸들러 선언의 일반 형식은 다음과 같습니다.

```
DECLARE handler-type  HANDLER FOR condition SQL-procedure-statement
```

DB2에서 *condition*과 일치하는 *조건*을 일으키면 DB2는 제어를 *조건* 핸들러로 전달합니다. 조건 핸들러는 *handler-type*에 의해 표시된 작업을 수행한 다음 *SQL-procedure-statement*를 실행합니다.

handler-type

CONTINUE

*SQL-procedure-statement*가 완료된 후 오류를 일으킨 명령문 다음 명령문으로 계속 실행함을 지정합니다.

EXIT *SQL-procedure-statement*가 완료된 후 핸들러가 들어 있는 복합 텍스트 명령문 끝에서 계속 실행함을 지정합니다.

UNDO

*SQL-procedure-statement*가 실행하기 전 DB2가 핸들러가 들어 있는 복합 텍스트 명령문에서 발생하는 모든 SQL 조작을 구간 복원함을 지정합니다. *SQL-procedure-statement*가 완료된 후 핸들러가 들어 있는 복합 텍스트 명령문 끝에서 계속 실행됩니다.

주: ATOMIC 복합 텍스트 명령문에서 UNDO 핸들러만 선언할 수 있습니다.

조건

DB2는 세 가지 일반 조건을 제공합니다.

NOT FOUND

+100의 SQLCODE 또는 '02000'의 SQLSTATE를 초래하는 모든 조건을 식별합니다.

SQLEXCEPTION

부정적인 SQLCODE를 초래하는 모든 조건을 식별합니다.

SQLWARNING

(SQLWARN0가 'W')라는 경고 조건이나 +100이 아닌 긍정적 SQL 리턴 코드를 초래하는 모든 조건을 식별합니다.

DECLARE문을 사용하여 특정 SQLSTATE에 대해 자체 조건을 정의할 수도 있습니다. 자체 조건 정의에 대한 자세한 내용은 *SQL 참조서*를 참조하십시오.

SQL-procedure-statement

단일 SQL 프로시듀어 명령문을 사용하여 조건 핸들러의 동작을 정의할 수 있습니다. DB2는 BEGIN...END 블록으로 구분된 복합 텍스트 명령문을 단일 SQL 프로시듀어 명령문으로 승인합니다. 복합 텍스트 명령문을 사용하여 조건 핸들러의 동작을 정의하고 핸들러로 SQLSTATE 또는 SQLCODE 변수의 값을 유지하면, 변수의 값을 복합 텍스트 블록의 첫번째 명령문에 있는 지역 변수 또는 매개변수에 할당해야 합니다. 복합 텍스트 블록의 첫번째 명령

문이 SQLSTATE 또는 SQLCODE의 값을 지역 변수나 매개변수에 할당하지 않으면, SQLSTATE와 SQLCODE는 Db2에서 조건 핸들러를 호출하게 만든 값을 유지할 수 없습니다.

주: 조건 핸들러에서 또다른 조건 핸들러를 정의할 수 없습니다.

다음 예는 간단한 조건 핸들러를 보여줍니다.

예: CONTINUE 핸들러: 이 핸들러는 DB2가 NOT FOUND 조건을 발생시킬 때 1의 값을 *at_end* 지역 변수에 할당합니다. 그러면 DB2는 제어를 NOT FOUND 조건을 일으킨 명령문 다음의 명령문으로 전달합니다.

```
DECLARE not_found CONDITION FOR SQLSTATE '02000';
DECLARE CONTINUE HANDLER FOR not_found SET at_end=1;
```

예: EXIT 핸들러: 프로시듀어는 NO_TABLE을 SQLSTATE 42704 (*name*은 정의되지 않은 이름)의 조건 이름으로 선언합니다. NO_TABLE의 조건 핸들러는 `Table does not exist` 문자열을 OUT_BUFFER 출력 매개변수에 배치합니다. 그러면 핸들러는 SQL 프로시듀어에서 핸들러가 선언된 복합 텍스트 명령문을 종료하도록 만듭니다.

```
DECLARE NO_TABLE CONDITION FOR SQLSTATE '42704';
DECLARE EXIT HANDLER FOR NO_TABLE
   BEGIN
      SET OUT_BUFFER='Table does not exist';
   END
```

예: UNDO 핸들러: 프로시듀어는 먼저 SQLSTATE에 대해 이름을 정의하지 않고 SQLSTATE 42704의 UNDO 조건 핸들러를 선언합니다. 핸들러는 SQL 프로시듀어에서 현재 작업 단위를 구간 복원하고 `Table does not exist` 문자열을 OUT_BUFFER 출력 매개변수에 배치하며, 핸들러가 선언된 복합 텍스트 명령문을 종료하게 만듭니다.

```
DECLARE UNDO HANDLER FOR SQLSTATE '42704'
   BEGIN
      SET OUT_BUFFER='Table does not exist';
   END;
```

주: ATOMIC 복합 텍스트 명령문에서 UNDO 핸들러만 선언할 수 있습니다.

SIGNAL 및 RESIGNAL문

SIGNAL과 RESIGNAL문을 사용하여 특정 SQLSTATE를 명시적으로 일으킬 수 있습니다. SIGNAL과 RESIGNAL문의 SET MESSAGE_TEXT절을 사용하여 DB가 사용자 정의된 SQLSTATE에 대해 표시할 텍스트를 정의하십시오.

다음 예에서 SQL 프로시듀어 본문은 사용자 정의 SQLSTATE 72822의 조건 핸들러를 선언합니다. 프로시듀어가 SQLSTATE 72822를 일으키는 SIGNAL문을 실행하면 DB2는 조건 핸들러를 호출합니다. 조건 핸들러는 IF문을 사용하여 SQL 변수 *var*의 값을 테스트합니다. *var*이 OK이면, 핸들러는 SQLSTATE 값을 72623으로 재정의하고 문자열 리터럴을 SQLSTATE 72623에 연관된 텍스트에 할당합니다. *var*이 OK가 아니면, 핸들러는 SQLSTATE 값을 72319로 재정의하고 *var*의 값을 그 SQLSTATE에 연관된 텍스트로 할당합니다.

```
DECLARE EXIT CONDITION HANDLER FOR SQLSTATE '72822'
   BEGIN
       IF ( var = 'OK' )
          RESIGNAL '72623' SET MESSAGE_TEXT = 'Got SQLSTATE 72822';
       ELSE
          RESIGNAL '72319' SET MESSAGE_TEXT = var;
   END;

SIGNAL SQLSTATE '72822';
```

SIGNAL 및 RESIGNAL문에 대한 자세한 정보는 *SQL 참조서*를 참조하십시오.

SQL 프로시듀어의 SQLCODE 및 SQLSTATE 변수

SQL 프로시듀어를 디버그하려면 SQL 프로시듀어의 다양한 위치에서 SQLSTATE와 SQLCODE의 값을 삽입하거나 SQLCODE와 SQLSTATE 값을 진단 문자열에서 OUT 매개변수로 리턴할 수 있습니다. SQLCODE와 SQLSTATE 값을 사용하려면 SQL 프로시듀어 본문에서 다음 SQl 변수를 선언하십시오.

```
DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
```

CONTINUE 조건 핸들러를 사용하여 SQLSTATE와 SQLCODE 변수의 값을 SQL 프로시듀어 본문에 있는 지역 변수에 지정할 수도 있습니다. 이런 지역 변수를 사용하여 절차적 논리를 제어하거나 값을 다시 출력 매개변수로 전달할 수 있

습니다. 다음 예에서 SQL 프로시듀어는 SQLCODE가 RETCODE라는 지역 변수에 설정된 각 SQL문 다음 명령문으로 제어를 리턴합니다.

```
DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE retcode INTEGER DEFAULT 0;

DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET retcode = SQLCODE;
DECLARE CONTINUE HANDLER FOR SQLWARNING SET retcode = SQLCODE;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET retcode = SQLCODE;
```

주: SQL 프로시듀어에서 SQLCODE나 SQLSTATE 변수를 액세스할 때 DB2는 후속 명령문에 대해 SQLCODE의 값을 0으로 설정하고 SQLSTATE의 값을 ‘00000’으로 설정합니다.

SQL 프로시듀어에서 동적 SQL 사용

외부 저장 프로시듀어와 같이 SQL 프로시듀어는 동적 SQL문을 실행할 수 있습니다. 동적 SQL문에 매개변수 표시문자가 없고 이를 한번만 실행할 계획이라면 EXECUTE IMMEDIATE문을 사용하십시오.

동적 SQL문에 매개변수 표시문자가 있으면 PREPARE와 EXECUTE문을 사용해야 합니다. 동적 SQL문을 여러 번 실행할 계획이라면 단일 PREPARE문을 실행하고 EXECUTE문을 여러 번 실행하는 것이 매번 EXECUTE IMMEDIATE문을 실행하는 것보다 더 효율적일 것입니다. PREPARE와 EXECUTE문을 사용하여 SQL 프로시듀어에서 동적 SQL을 실행하려면 다음 명령문을 SQL 프로시듀어 본문에 포함시켜야 합니다.

단계 1. DECLARE문을 사용하여 동적 SQL을 보관할 만큼 큰 유형 VARCHAR의 변수를 선언하십시오.

단계 2. SET문을 사용하여 명령문 문자열을 변수에 할당하십시오. 명령문 문자열에 직접 변수를 포함시킬 수 없습니다. 대신 물음표('?') 기호를 명령문에서 사용된 모든 변수의 매개변수 표시문자로 사용해야 합니다.

단계 3. PREPARE문을 사용하여 명령문 문자열에서 준비된 명령문을 작성합니다.

단계 4. EXECUTE문을 사용하여 준비된 명령문을 실행합니다. 명령문 문자열에 매개변수 표시문자가 들어 있으면 USING절을 사용하여 이를 변수 값으로 바꾸십시오.

주: SQL 프로시듀어에 대해 PREPARE문에 정의된 명령문 이름은 범위가 정해진 변수로 처리됩니다. 일단 SQL 프로시듀어가 명령문 이름을 정의할 수 있는 범위에 존재하면 DB2는 더 이상 명령문 이름을 액세스하지 않습니다. 복합 텍스트 명령문에서 같은 명령문 이름을 사용하는 PREPARE문을 두 개 발행할 수 없습니다.

예: 동적 SQL문: 다음 예는 동적 SQL문이 들어 있는 SQL 프로시듀어를 보여줍니다.

프로시듀어는 부서 번호(*deptNumber*)를 입력 매개변수로 수신합니다. 프로시듀어에서 3개의 명령문 문자열이 구축되고 준비되며 실행됩니다. 첫번째 명령문 문자열은 DROP문을 실행하여 작성해야 하는 테이블이 아직 없음을 확인합니다. 이 테이블은 DEPT_*deptno*_T로 명명됩니다. 여기서 *deptno*는 *deptNumber* 입력 매개변수의 값입니다. CONTINUE HANDLER를 통해 SQL 프로시듀어는 테이블이 없는 경우 DROP문에서 DB2가 리턴하는 SQLSTATE 42704("정의되지 않은 오브젝트 이름")를 가지할 때 계속할 수 있게 합니다. 둘째 명령문 문자열은 CREATE 문을 실행하여 DEPT_*deptno*_T를 작성합니다. 셋째 명령문 문자열은 *deptno* 부서에 있는 작원에 대한 행을 DEPT_*deptno*_T에 삽입합니다. 셋째 명령문 문자열에는 *deptNumber*을 나타내는 매개변수 표시문자가 있습니다. 준비된 명령문이 실행될 때 *deptNumber* 매개변수는 매개변수 표시문자에 대해 대체됩니다.

```
CREATE PROCEDURE create_dept_table
(IN deptNumber VARCHAR(3), OUT table_name VARCHAR(30))
LANGUAGE SQL
  BEGIN
    DECLARE stmt VARCHAR(1000);

    -- continue if sqlstate 42704 ('undefined object name')
    DECLARE CONTINUE HANDLER FOR SQLSTATE '42704'
      SET stmt = '';
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
      SET table_name = 'PROCEDURE_FAILED';

    SET table_name = 'DEPT_'||deptNumber||'_T';
    SET stmt = 'DROP TABLE '||table_name;
    PREPARE s1 FROM stmt;
    EXECUTE s1;
    SET stmt = 'CREATE TABLE '||table_name||
     '( empno CHAR(6) NOT NULL, '||
     'firstnme VARCHAR(12) NOT NULL, '||
```

```
     'midinit CHAR(1) NOT NULL, '||
     'lastname VARCHAR(15) NOT NULL, '||
     'salary DECIMAL(9,2))';
    PREPARE s2 FROM STMT;
    EXECUTE s2;
    SET stmt = 'INSERT INTO '||table_name || ' ' ||
     'SELECT empno, firstnme, midinit, lastname, salary '||
     'FROM employee '||
     'WHERE workdept = ?';
   PREPARE s3 FROM stmt;
   EXECUTE s3 USING deptNumber;
END
```

중첩 SQL 프로시듀어

SQL 프로시듀어에는 다른 SQL 프로시듀어를 호출하는 CALL문이 있습니다. 이 기능은 *중첩* 저장 프로시듀어라고 하는데, 기존 SQL 프로시듀어를 다시 사용하여 좀 더 복잡한 응용프로그램을 설계할 수 있습니다.

중첩 SQL 프로시듀어간의 매개변수 전달

호출자 SQL 프로시듀어에서 목표 SQL 프로시듀어를 호출하려면 적합한 번호와 매개변수 유형의 CALL문을 호출자에 포함시키기만 하면 됩니다. 목표가 OUT 매개변수를 리턴하면 호출자는 리턴된 값을 자체 명령문에서 사용할 수 있습니다.

예를 들어, 목표 SQL 프로시듀어를 "SALES_TARGET" 이름과 함께 호출하고 다음 SQL을 사용하여 INTEGER 유형의 단일 OUT 매개변수를 승인하는 SQL 프로시듀어를 작성할 수 있습니다.

```
CREATE PROCEDURE NEST_SALES(OUT budget DECIMAL(11,2))
LANGUAGE SQL
BEGIN
   DECLARE total INTEGER DEFAULT 0;
   SET total = 6;
   CALL SALES_TARGET(total);
   SET budget = total * 10000;
END
```

중첩 SQL 프로시듀어에서 결과 세트 리턴

목표 SQL 프로시듀어가 결과 세트를 리턴하면 목표 SQL 프로시듀어에서 발행된 DECLARE CURSOR문에 따라 호출자 또는 클라이언트 응용프로그램이 결과 세

트를 수신합니다. 목표에 있으면서 WITH RETURN TO CLIENT절을 포함하는 각 DECLARE CURSOR문에 대해 호출자는 결과 세트를 수신하지 않습니다. WITH RETURN TO CLIENT 커서에 대해 결과 세트는 직접 클라이언트 응용 프로그램에 리턴됩니다.

중첩 SQL 프로시듀어에서 결과 세트를 리턴하는 것에 대한 자세한 내용은 276 페이지의 『클라이언트 및 호출자 결과 세트 리턴』을 참조하십시오.

중첩 SQL 프로시듀어에 제한사항

응용프로그램 구조를 설계할 때 다음 제한사항을 염두에 두십시오.

LANGUAGE
: SQL 프로시듀어는 SQL이나 C로 작성된 저장 프로시듀어만 호출할 수 있습니다. SQL 프로시듀어에서 다른 호스트 언어 저장 프로시듀어를 호출할 수 없습니다.

16 레벨의 중첩
: SQL 프로시듀어에 중첩 호출을 최대 16 레벨까지 포함시킬 수 있습니다. SQL 프로시듀어 A가 SQL 프로시듀어 B를 호출하고 SQL 프로시듀어 B가 SQL 프로시듀어 C를 호출하는 시나리오는 3 레벨의 중첩 호출의 예입니다.

반복
: 스스로를 반복적으로 호출하는 SQL 프로시듀어를 작성할 수 있습니다. 반복 SQL 프로시듀어는 최대 중첩 레벨에 대해 앞에서 설명한 제한사항을 준수해야 합니다.

보안
: SQL 프로시듀어는 더 높은 SQL 데이터 액세스 레벨로 카탈로그화되는 목표 SQL 프로시듀어를 호출할 수 없습니다. 예를 들어, CONTAINS SQL절로 작성된 SQL 프로시듀어는 CONTAINS SQL절이나 NO SQL 절로 작성된 SQL 프로시듀어를 호출할 수 있으며, READS SQL DATA 절이나 MODIFIES SQL DATA절로 작성된 SQL 프로시듀어를 호출할 수 없습니다.

  NO CLAUSE절로 작성된 SQL 프로시듀어는 CALL문을 발행합니다.

SQL 프로시듀어로부터 결과 세트 리턴

SQL 프로시듀어에서 결과 세트를 리턴하는 것은 외부 저장 프로시듀어에서 결과 세트를 리턴하는 것과 비슷합니다. 클라이언트 응용프로그램은 CLI, JDBC 또는 SQLJ 응용프로그램 인터페이스를 사용하여 SQL 프로시듀어에서 결과 세트를 승인해야 합니다. 다른 SQL 프로시듀어를 호출하는 SQL 프로시듀어 또한 이런 프로시듀어에서 결과 세트를 승인해야 합니다. SQL 프로시듀어에서 결과 세트를 리턴하려면 다음과 같이 SQL 프로시듀어를 작성하십시오.

1. CREATE PROCEDURE문의 DYNAMIC RESULT SETS절을 사용하여 SQL 프로시듀어가 리턴하는 결과 세트의 수를 선언하십시오.
2. DECLARE CURSOR문을 사용하여 커서를 선언하십시오.
3. OPEN CURSOR문을 사용하여 커서를 여십시오.
4. 커서를 닫지 않고 SQL 프로시듀어를 종료하십시오.

예를 들어, 다음과 같이 *threshold* INOUT 매개변수의 값을 기반으로 단일 결과 세트를 리턴하는 SQL 프로시듀어를 작성할 수 있습니다.

```
CREATE PROCEDURE RESULT_SET (INOUT threshold SMALLINT)
LANGUAGE SQL
DYNAMIC RESULT SETS 1
BEGIN
   DECLARE cur1 CURSOR WITH RETURN TO CALLER FOR
       SELECT name, job, years
       FROM staff
       WHERE years < threshold;
   OPEN cur1;
END
```

클라이언트 및 호출자 결과 세트 리턴

응용프로그램이 중첩 SQL 프로시듀어에서 결과 세트를 리턴하면, DECLARE CURSOR문의 WITH RETURN절을 사용하여 DB2가 결과 세트를 적합한 위치로 리턴하게 해야 합니다. 목표 SQL 프로시듀어가 결과 세트를 호출하는 SQL 프로시듀어로 리턴하면 호출자는 ALLOCATE CURSOR와 ASSOCIATE RESULT SET LOCATOR문을 사용하여 결과 세트를 액세스하고 사용해야 합니다.

클라이언트 결과 세트 리턴

항상 SQL 프로시듀어에서 클라이언트 응용프로그램으로 결과 세트를 리턴하려면, 결과 세트에 연관된 DECLARE CURSOR문의 WITH RETURN TO CLIENT 절을 사용하십시오. 다음 예에서 "CLIENT_SET"가 중첩 SQL 프로시듀어 CALL 문의 목표인 경우라도 SQL 프로시듀어 "CLIENT_SET"는 DECLARE CURSOR 문의 RETURN TO CLIENT절을 사용하여 결과 세트를 클라이언트 응용프로그램으로 리턴합니다.

```
CREATE PROCEDURE CLIENT_SET()
DYNAMIC RESULT SETS 1
LANGUAGE SQL
BEGIN
   DECLARE clientcur CURSOR WITH RETURN TO CLIENT
       FOR SELECT name, dept, job
       FROM staff
       WHERE salary > 20000;
   OPEN clientcur;
END
```

호출자 결과 세트 리턴

호출자가 클라이언트 응용프로그램이든 다른 SQL 프로시듀어든지 간에 결과 세트를 SQL 프로시듀어의 직접 호출자에게 리턴하려면, 결과 세트에 연관된 DECLARE CURSOR문의 WITH RETURN TO CALLER절을 사용하십시오. 다음 예에서 SQL 프로시듀어 "CALLER_SET"는 WITH RETURN TO CALLER절을 사용하여 결과 세트를 CALLER_SET 호출자에게 리턴합니다.

```
CREATE PROCEDURE CALLER_SET()
DYNAMIC RESULT SETS 1
LANGUAGE SQL
BEGIN
   DECLARE clientcur CURSOR WITH RETURN TO CALLER
       FOR SELECT name, dept, job
       FROM staff
       WHERE salary > 15000;
   OPEN clientcur;
END
```

호출자 결과 세트 리턴

호출하는 SQL 프로시듀어가 목표 SQL 프로시듀어에서 결과 세트를 수신할 것으로 예상될 때, ALLOCATE CURSOR와 ASSOCIATE RESULT SET LOCATOR문을 사용하여 결과 세트를 액세스하고 사용해야 합니다.

ASSOCIATE RESULT SET LOCATOR

호출자에게 결과 세트를 하나 이상 리턴하는 목표 SQL 프로시듀어에 대한 CALL문 다음에 호출하는 SQL 프로시듀어는 이 명령문을 발행하여 결과 세트 위치 지정자 변수를 리턴된 각 결과 세트마다 지정해야 합니다. 예를 들어, 목표 SQL 프로시듀어에서 세 개의 결과 세트를 수신할 것으로 예상되는 호출하는 SQL 프로시듀어에는 다음과 같은 SQL이 들어 있습니다.

```
DECLARE result1 RESULT_SET_LOCATOR VARYING;
DECLARE result2 RESULT_SET_LOCATOR VARYING;
DECLARE result3 RESULT_SET_LOCATOR VARYING;

CALL targetProcedure();
ASSOCIATE RESULT SET LOCATORS(result1, result2, result3)
   WITH PROCEDURE targetProcedure;
```

ALLOCATE CURSOR

호출하는 SQL 프로시듀어에서 ALLOCATE CURSOR문을 사용하여 목표 SQL 프로시듀어에서 리턴된 결과 세트를 여십시오. ALLOCATE CURSOR문을 사용하려면 결과 세트는 항상 ASSOCIATE RESULT SET LOCATORS문을 통해 결과 세트 위치 지정자에 연관되어야 합니다. 일단 SQL 프로시듀어가 ALLOCATE CURSOR문을 발행하면, ALLOCATE CURSOR문에서 선언된 커서 이름을 사용하여 결과 세트에서 행을 페치할 수 있습니다. 이전에 설명된 ASSOCIATE LOCATORS 예를 확장하려면, SQL 프로시듀어는 다음 SQL을 사용하여 리턴된 결과 세트 중 첫번째 세트에서 행을 페치할 수 있습니다.

```
DECLARE result1 RESULT_SET_LOCATOR VARYING;
DECLARE result2 RESULT_SET_LOCATOR VARYING;
DECLARE result3 RESULT_SET_LOCATOR VARYING;
CALL targetProcedure();
ASSOCIATE RESULT SET LOCATORS(result1, result2, result3)
   WITH PROCEDURE targetProcedure;
ALLOCATE rsCur CURSOR FOR result1;
WHILE (at_end = 0) DO
   SET total1 = total1 + var1;
   SET total2 = total2 + var2;
   FETCH FROM rsCur INTO var1, var2;
END WHILE;
```

SQL 프로시듀어 디버깅

SQL 프로시듀어를 작성한 다음 267 페이지의 『CREATE PROCEDURE문 실행』에 설명된 대로 CREATE PROCEDURE문을 발행해야 합니다. 특정한 경우 DB2는 CREATE PROCEDURE문의 응답으로 오류를 리턴합니다. 오류를 정정하는 것에 대한 설명과 제안을 포함하여 DB2에서 리턴된 오류에 대해 더 많은 정보를 검색하려면 CLP에서 다음 명령을 발행하십시오.

```
db2 "? error-code"
```

여기서 *error-code*는 오류에 의해 리턴된 SQLCODE 또는 SQLSTATE를 나타냅니다. 예를 들어, CREATE PROCEDURE문이 SQLCODE "SQL0469N"("매개변수 모드가 유효하지 않음")와 함께 오류를 리턴하면 다음 명령을 발행하십시오.

```
db2 "? SQL0469"
```

DB2는 다음 메시지를 리턴합니다.

```
Explanation:  One of the following errors occurred:

o   a parameter in an SQL procedure is declared as OUT and is
    used as input in the procedure body

o   a parameter in an SQL procedure is declared as IN and is
    modified in the procedure body.

User Response:  Change the attribute of the parameter to INOUT,
or change the use of the parameter within the procedure.
```

일단 메시지를 표시하면 "사용자 조치" 절에 있는 제안에 따라 SQL 프로시듀어를 수정하십시오.

SQL 프로시듀어 오류 메시지 표시

SQL 프로시듀어에 대해 CREATE PROCEDURE문을 발행하면 DB2는 SQL 프로시듀어 본문의 구문을 승인하여 사전 처리 컴파일이나 컴파일 단계에서 SQL 프로시듀어를 작성할 수 있습니다. 이런 상황에서 DB2는 보통 오류 메시지가 들어 있는 로그 파일을 작성합니다. 이 로그 파일과 기타 중단 파일에 대해서는 282 페이지의 『중간 파일을 사용한 SQL 프로시듀어 디버깅』에 설명되어 있습니다.

SQL 프로시듀어에 대해 DB2와 C 컴파일러에서 생성된 오류 메시지를 검색하려면 데이터베이스 서버에서 다음 디렉토리에 메시지 로그 파일을 표시하십시오.

UNIX *$DB2PATH*/function/routine/sqlproc/*$DATABASE*/*$SCHEMA*/tmp

여기서 *$DB2PATH*는 인스턴스 디렉토리의 위치를 나타내고 *$DATABASE*는 데이터베이스 이름을 나타내며 *$SCHEMA*는 SQL 프로시듀어를 작성할 때 사용되는 스키마 이름을 나타냅니다.

Windows NT

%DB2PATH%\function\routine\sqlproc*%DB%**%SCHEMA%*\tmp

여기서 *%DB2PATH%*는 인스턴스 디렉토리의 위치를 나타내고 *%DB%*는 데이터베이스 이름을 나타내며 *%SCHEMA%*는 SQL 프로시듀어를 작성할 때 사용되는 스키마 이름을 나타냅니다.

응용프로그램에서 CALL문을 발행하여 다음 구문을 통해 db2udp!get_error_messages 샘플 저장 프로시듀어를 호출할 수도 있습니다.

```
CALL db2udp!get_error_messages(schema-name, file-name, message-text)
```

여기서 *schema-name*은 SQL 프로시듀어의 스키마를 나타내는 입력 매개변수이고 *file-name*은 SQL 프로시듀어에 대해 생성된 파일 이름이며 *message-text*는 메시지 로그 파일로 메시지 텍스트를 리턴하는 출력 매개변수입니다.

예를 들어, 다음 Java 응용프로그램을 사용하여 SQL 프로시듀어에 대해 오류 메시지를 표시할 수 있습니다.

```
public static String getErrorMessages(Connection  con,
  String procschema, String filename) throws Exception
{
    String filecontents = null;
    // prepare the CALL statement
    CallableStatement stmt = null;
    try
    {
        String sql = "Call db2udp!get_error_messages(?, ?, ?) ";
        stmt = con.prepareCall (sql);

        // set all parameters (input and output)
        stmt.registerOutParameter( 3, java.sql.Types.LONGVARCHAR  );
        stmt.setString( 1, procschema );
        stmt.setString( 2, filename );

        // call the stored procedure
        boolean isrs = stmt.execute();
```

```
            filecontents = stmt.getString(3);

            System.out.println("SQL Procedure - getErrorMessages "
              + filecontents);
            return filecontents;
        }
        catch (Exception e) { throw e; }
        finally
        {
            if (stmt != null) stmt.close();
        }
    }
```

다음 C 응용프로그램을 사용하여 SQL 프로시듀어에 대해 오류 메시지를 표시할 수 있습니다.

```
int getErrors(char inputSchema[9], char inputFilename[9],
                         char outputFilecontents[32000])
{
  EXEC SQL BEGIN DECLARE SECTION;
    char    procschema[100] = "";
    char    filename[100] = "";
    char    filecontents[32000] = "";
  EXEC SQL END DECLARE SECTION;

  strcpy (procschema, inputSchema);
  strcpy (filename, inputFilename);

  EXEC SQL CALL "db2udp!get_error_messages"
    (:procschema, :filename, :filecontents);
  if ( sqlca.sqlcode != 0 )
  {
    printf("Call failed. Code: %d\n", sqlca.sqlcode);
    return 1;
  }
  else
  {
    printf("\nSQL Procedure - getErrors:\n%s\n", filecontents);
  }
  strcpy (outputFilecontents, filecontents);
  return 0;
}
```

주: DB2에서 작성하지 못한 SQL 프로시듀어에 대해 오류 메시지를 표시하기 전에 프로시듀어 이름과 SQL 프로시듀어의 생성된 파일 이름을 알고 있어야 합니다. 프로시듀어 스키마 이름이 CREATE PROCEDURE문의 일부로 발행

되지 않으면 DB2는 CURRENT SCHEMA 특수 레지스터의 값을 사용합니다. CURRENT SCHEMA 특수 레지스터의 값을 표시하려면 CLP에서 다음 명령문을 발행하십시오.

```
VALUES CURRENT SCHEMA
```

중간 파일을 사용한 SQL 프로시듀어 디버깅

SQL 프로시듀어에 대해 CREATE PROCEDURE문을 발행하고 DB2가 SQL 프로시듀어 본문의 구문을 승인하면 DB2는 여러 개의 중간 파일을 사용하여 SQL 프로시듀어를 작성합니다. DB2가 성공적으로 SQL 프로시듀어를 작성한 다음 이는 보통 중간 파일을 제거하여 시스템 자원을 절약합니다. DB2가 CREATE PROCEDURE 구문을 승인하지만 SQL 프로시듀어를 작성하지 못하면 이는 CREATE PROCEDURE 프로세스의 사전 처리 컴파일, 바인드 그리고 컴파일 단계를 추적하는 로그 파일을 유지합니다.

UNIX 시스템에서 DB2는 *instance*/function/routine/sqlproc/*dbAlias*/*schema*의 기본 디렉토리를 사용하여 중간 파일을 보관합니다. 여기서 *instance*는 DB2 인스턴스의 경로를 나타내고 *dbAlias*는 데이터베이스 별명을 나타내며 *schema*는 CREATE PROCEDURE문이 발행된 스키마를 나타냅니다.

OS/2와 Windows 32-비트 운영 체제에서 DB2는 *instance*\function\routine\sqlproc*dbAlias**schema*의 기본 디렉토리를 사용하여 중간 파일을 보관합니다. 여기서 *instance*는 DB2 인스턴스를 나타내고 *dbAlias*는 데이터베이스 별명을 나타내며 *schema*는 CREATE PROCEDURE문이 발행된 스키마를 나타냅니다.

SQL 프로시듀어가 성공적으로 작성되었지만 CALL문에서 예상 결과를 리턴하지 않으면 중간 파일을 검사할 수 있습니다. DB2가 중간 파일을 제거하지 못하게 막으려면 다음 명령을 사용하여 DB2_SQLROUTINE_KEEP_FILES DB2 레지스트리 변수를 "예"로 설정하십시오.

```
db2set DB2_SQLROUTINE_KEEP_FILES="yes"
```

DB2가 레지스트리 변수의 새 값을 사용하기 전에 데이터베이스를 다시 시작해야 합니다.

SQL 프로시듀어의 예

이 절에는 SQL 프로시듀어 본문에 나타나는 각 명령문을 사용하는 방법에 대한 예가 있습니다. SQL 프로시듀어를 호출하는 클라이언트 응용프로그램을 포함하여 이런 예와 기타 SQL 프로시듀어에 대해 다음 디렉토리를 참조하십시오.

UNIX 운영 체제

*$HOME*/sqllib/samples/sqlproc, 여기서 *$HOME*은 DB2 인스턴스 디렉토리의 위치를 나타냅니다.

Windows 32 비트 운영 체제

%DRIVE%\sqllib\samples\sqlproc, 여기서 *%DRIVE%*는 DB2를 설치한 드라이브를 나타냅니다.

예 1: CASE문: 다음 SQL 프로시듀어는 CASE문을 사용하는 방법을 보여줍니다. 프로시듀어는 직원의 ID 번호와 등급을 입력 매개변수로 받습니다. CASE문은 가능한 등급마다 다른 UPDATE문을 사용하여 직원에 대해 급여와 보너스를 수정합니다.

```
CREATE PROCEDURE UPDATE_SALARY
(IN employee_number CHAR(6), IN rating INT)
LANGUAGE SQL
BEGIN
  DECLARE not_found CONDITION FOR SQLSTATE '02000';
  DECLARE EXIT HANDLER FOR not_found
    SIGNAL SQLSTATE '02444';

  CASE rating
    WHEN 1 THEN
      UPDATE employee
      SET salary = salary * 1.10, bonus = 1000
      WHERE empno = employee_number;
    WHEN 2 THEN
      UPDATE employee
      SET salary = salary * 1.05, bonus = 500
      WHERE empno = employee_number;
    ELSE
      UPDATE employee
      SET salary = salary * 1.03, bonus = 0
      WHERE empno = employee_number;
   END CASE;
END
```

예 2: 중첩 IF 및 WHILE문의 복합 텍스트 명령문: 다음 예는 중첩 IF문, WHILE 문 및 지정 명령문이 들어 있는 복합 텍스트 명령문을 보여줍니다. 예는 SQL 변수, 커서 그리고 오류 코드 클래스에 대한 핸들러도 보여줍니다.

프로시듀어는 부서 번호를 입력 매개변수로 수신합니다. 프로시듀어 본문에 있는 WHILE문은 부서의 각 직원에 대해 급여와 보너스를 패치합니다. WHILE문에 있는 IF문이 일한 연수와 현재 급여에 따라 각 직원에 대해 급여를 갱신합니다. 부서에 있는 모든 직원 레코드가 처리되면 직원 레코드를 검색하는 FETCH문은 SQLSTATE 20000을 수신합니다. *not_found* 조건 핸들러는 WHILE문에 대해 검색 조건을 거짓으로 만드므로 WHILE문의 실행이 종료됩니다.

```
CREATE PROCEDURE BUMP_SALARY_IF (IN deptnumber SMALLINT)
LANGUAGE SQL
BEGIN
   DECLARE v_salary DOUBLE;
   DECLARE v_years SMALLINT;
   DECLARE v_id SMALLINT;
   DECLARE at_end INT DEFAULT 0;
   DECLARE not_found CONDITION FOR SQLSTATE '02000';

   -- CAST salary as DOUBLE because SQL procedures do not support DECIMAL
   DECLARE C1 CURSOR FOR
     SELECT id, CAST(salary AS DOUBLE), years
     FROM staff;
   DECLARE CONTINUE HANDLER FOR not_found
     SET at_end = 1;

   OPEN C1;
   FETCH C1 INTO v_id, v_salary, v_years;
   WHILE at_end = 0 DO
     IF (v_salary < 2000 * v_years)
       THEN UPDATE staff
         SET salary = 2150 * v_years
         WHERE id = v_id;
       ELSEIF (v_salary < 5000 * v_years)
         THEN IF (v_salary < 3000 * v_years)
           THEN UPDATE staff
             SET salary = 3000 * v_years
             WHERE id = v_id;
         ELSE UPDATE staff
           SET salary = v_salary * 1.10
           WHERE id = v_id;
         END IF;
       ELSE UPDATE staff
         SET job = 'PREZ'
         WHERE id = v_id;
     END IF;
     FETCH C1 INTO v_id, v_salary, v_years;
   END WHILE;
   CLOSE C1;
END
```

예 3: 전역 임시 테이블 및 결과 세트가 있는 중첩 SQL 프로시듀어 사용:

다음 예는 ASSOCIATE RESULT SET LOCATOR와 ALLOCATE CURSOR 문을 사용하여 호출된 SQL 프로시듀어인 temp_table_insert에서 결과 세트를 호출하는 SQL 프로시듀어인 temp_table_create로 리턴하는 방법을 보여줍니다. 예는 또한 호출된 SQL 프로시듀어가 호출하는 SQL 프로시듀어에 의해 작성된 전역 임시 테이블을 사용하는 방법을 보여줍니다.

예에서 클라이언트 응용프로그램 또는 다른 SQL 프로시듀어는 SESSION.TTT 전역 임시 테이블을 작성하고, temp_table_insert를 호출하는 temp_table_create를 호출합니다.

SESSION.TTT 전역 임시 테이블을 사용하기 위해 temp_table_insert에는 temp_table_create가 SESSION.TTT를 작성하기 위해 발행한 명령문과 같은 DECLARE GLOBAL TEMPORARY TABLE문이 들어 있습니다. 차이점은 temp_table_insert에 DECLARE GLOBAL TEMPORARY TABLE문에 항상 거짓인 IF문이 있다는 것입니다. IF문은 DB2가 다시 전역 임시 테이블을 작성하지 못하도록 막지만, SQL 프로시듀어에서 전역 임시 테이블을 후속 명령문에서 사용할 수 있게 합니다.

다른 SQL 프로시듀어에 의해 작성된 전역 임시 테이블에서 결과 세트를 리턴하려면, temp_table_insert는 새 범위에서 DECLARE CURSOR문을 발행해야 합니다. temp_table_insert는 새 범위의 요구사항을 만족시키는 복합 SQL 블록에서 DECLARE CURSOR와 OPEN CURSOR문을 발행합니다. 커서는 SQL 프로시듀어가 존재하기 전에는 닫히지 않으므로, DB2는 결과 세트를 다시 호출자 temp_table_create로 전달합니다.

호출된 SQL 프로시듀어에서 결과 세트를 승인하려면 temp_table_create는 temp_table_insert를 결과 세트의 개시자로 식별하는 ASSOCIATE RESULT SET LOCATOR문을 발행합니다. 그러면 temp_table_create는 결과 세트를 열기 위해 결과 세트 위치 지정자에 대해 ALLOCATE CURSOR문을 발행합니다. ALLOCATE CURSOR문이 성공하면 SQL 프로시듀어는 보통 때처럼 결과 세트에 대해 작업할 수 있습니다. 이 예에서 temp_table_create는 결과 세트에서 모든 행을 페치하여 컬럼의 값을 출력 매개변수에 추가합니다.

주: 전역 임시 테이블을 사용하는 SQL 프로시듀어에 대해 CREATE PROCEDURE문을 발행하기 전에 사용자 임시 테이블 공간을 작성해야 합니다. 사용자 임시 테이블 공간을 작성하려면 다음 SQL문을 발행하십시오.

```
CREATE USER TEMPORARY TABLESPACE ts1
  MANAGED BY SYSTEM USING ('ts1file');
```

여기서 *ts1*은 사용자 임시 테이블 공간의 이름을 나타내고 *ts1file*은 테이블 공간에서 사용된 컨테이너의 이름을 나타냅니다.

```
CREATE PROCEDURE temp_table_create(IN parm1 INTEGER, IN parm2 INTEGER,
   OUT parm3 INTEGER, OUT parm4 INTEGER)
   LANGUAGE SQL
   BEGIN
      DECLARE loc1 RESULT_SET_LOCATOR VARYING;
      DECLARE total3,total4 INTEGER DEFAULT 0;
      DECLARE rcolumn1, rcolumn2 INTEGER DEFAULT 0;
      DECLARE result_set_end INTEGER DEFAULT 0;
      DECLARE CONTINUE HANDLER FOR NOT FOUND, SQLEXCEPTION, SQLWARNING
      BEGIN
        SET result_set_end = 1;
      END;
      --Create the temporary table that is used in both this SQL procedure
      --and in the SQL procedure called by this SQL procedure.
      DECLARE GLOBAL TEMPORARY TABLE ttt(column1 INT, column2 INT)
         NOT LOGGED;
      --Insert rows into the temporary table.
      --The result set includes these rows.
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+1, parm2+1);
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+2, parm2+2);
      --Make a nested call to the 'temp_table_insert' SQL procedure.
      CALL temp_table_insert(parm1, parm2);
      --Issue the ASSOCIATE RESULT SET LOCATOR statement to
      --accept a single result set from 'temp_table_insert'.
      --If 'temp_table_insert' returns multiple result sets,
      --you must declare one locator variable (for example,
      --ASSOCIATE RESULT SET LOCATOR(loc1, loc2, loc3) for each result set.
      ASSOCIATE RESULT SET LOCATOR(loc1) WITH PROCEDURE temp_table_insert;
      --The ALLOCATE statement is similar to the OPEN statement.
      --It makes the result set available in this SQL procedure.
      ALLOCATE cursor1 CURSOR FOR RESULT SET loc1;
      --Insert rows into the temporary table.
      --The result set does not include these rows.
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+5, parm2+5);
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+6, parm2+6);
      SET result_set_end = 0;
      --Fetch the columns from the first row of the result set.
      FETCH FROM cursor1 INTO rcolumn1, rcolumn2;
      WHILE (result_set_end = 0) DO
        SET total3 = total3 + rcolumn1;
        SET total4 = total4 + rcolumn2;
        --Fetch columns from the result set for the
        --next iteration of the WHILE loop.
        FETCH FROM cursor1 INTO rcolumn1, rcolumn2;
      END WHILE;
      CLOSE cursor1;
      SET parm3 = total3;
      SET parm4 = total4;
```

```
   END @

CREATE PROCEDURE temp_table_insert (IN parm1 INTEGER, IN parm2 INTEGER )
   LANGUAGE SQL
   BEGIN
      DECLARE result_set_end INTEGER DEFAULT 0;
      DECLARE CONTINUE HANDLER FOR NOT FOUND BEGIN
        SET result_set_end = 1;
      END;
      --To use a temporary table that is created by a different stored
      --procedure, include a DECLARE GLOBAL TEMPORARY TABLE statement
      --inside a condition statement that always evaluates to false.
      IF (1 = 0) THEN
        DECLARE GLOBAL TEMPORARY TABLE ttt(column1 INT, column2 INT)
           NOT LOGGED;
      END IF;
      --Insert rows into the temporary table.
      --The result set includes these rows.
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+3, parm2+3);
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+4, parm2+4);
      --To return a result set from the temporary table, issue
      --the DECLARE CURSOR statement inside a new scope, such as
      --a compound SQL statement (BEGIN...END block).
      --Issue the DECLARE CURSOR statement after the DECLARE
      --GLOBAL TEMPORARY TABLE statement.
      BEGIN
         --The WITH RETURN TO CALLER clause causes the SQL procedure
         --to return its result set to the calling procedure.
         DECLARE cur1 CURSOR WITH RETURN TO CALLER
            FOR SELECT * FROM session.ttt;
         --To return a result set, open a cursor without closing the cursor.
        OPEN cur1 ;
      END;
   END
```

제9장 IBM DB2 Stored Procedure Builder

Stored Procedure Builder란?. 289
Stored Procedure Builder 사용 이점 . . . 291
새로운 저장 프로시듀어 작성 291
기존의 저장 프로시듀어 작업 291
Stored Procedure Builder 프로젝트 작성 292
저장 프로시듀어 디버깅. 292

Stored Procedure Builder란?

Stored Procedure Builder는 DB2 저장 프로시듀어의 신속한 개발을 지원하는 그래픽 응용프로그램입니다. Stored Procedure Builder를 사용하여 다음 타스크를 수행할 수 있습니다.

- 새로운 저장 프로시듀어 작성
- 지역 및 원격 DB2 서버에서 저장 프로시듀어 구축
- 기존의 저장 프로시듀어 수정 및 재구축
- 설치된 저장 프로시듀어 실행의 테스트 및 디버그

저장 프로시듀어가 있는 응용프로그램을 작성하려면 Stored Procedure Builder는 OS/2, OS/390, OS/400, AIX, HP-UX, Linux, Solaris 운영 환경 및 Windows 32-비트 운영 체제를 포함한 전체 DB2 Universal Database 계열을 지원하는 단일 개발 환경을 제공합니다.

Stored Procedure Builder에 대해 지원되는 플랫폼:

Stored Procedure Builder는 AIX, Solaris** 운영 환경** 및 Windows 32-비트 운영 체제에서 DB2 응용프로그램 개발 클라이언트의 선택적 구성요소입니다.

클라이언트에서 Stored Procedure Builder를 사용하여 다음과 같은 플랫폼에 대해 DB2 Universal Database 서버에서 Java 저장 프로시듀어 및 SQL 프로시듀어를 구축하고 전개할 수 있습니다.

언어 저장 프로시듀어	DB2 UDB 플랫폼 지원
Java	OS/2, OS/390, AIX, HP-UX, Linux, Solaris 운영 환경 및 Windows 32 비트 운영 체제

© Copyright IBM Corp. 1993, 2000

SQL	OS/2, OS/390, OS/400, AIX, HP-UX, Linux, Solaris 운영 환경 및 Windows 32비트 운영 체제

SQL 저장 프로시듀어를 내보내고 Java 저장 프로시듀어를 기존 Java 클래스 파일에서 작성할 수 있습니다. 편리한 개발 환경을 제공하려면 Stored Procedure Builder 코드 편집기는 기본 키 바인딩 외에 `vi` 또는 `emacs` 키 바인딩의 사용을 허용해야 합니다.

Stored Procedure Builder 시작:

Windows 32비트 운영 체제의 DB2 Universal Database 프로그램 그룹에서 `db2spb` 명령을 명령행이나 다음 개발 응용프로그램에서 실행하여 Stored Procedure Builder를 시작할 수 있습니다.

- Microsoft Visual C++ 5.0 및 6.0
- Microsoft Visual Basic 5.0 및 6.0
- IBM VisualAge for Java

AIX 및 Solaris 운영 환경 클라이언트에서 `db2spb` 명령을 명령행에서 실행하여 Stored Procedure Builder를 시작할 수 있습니다.

Stored Procedure Builder는 Java를 이용하여 구현되며, 모든 데이터베이스 연결성은 JDBC(Java Database Connectivity)에서 관리됩니다. JDBC 드라이버를 사용하여 지역 DB2 별명 또는 호스트, 포트 및 데이터베이스 이름을 지정할 수 있는 기타 데이터베이스에 연결할 수 있습니다.

주: Stored Procedure Builder를 사용하려면 개발용 DB2 데이터베이스에 연결해야 합니다. Stored Procedure Builder 사용에 대한 세부사항은 IBM DB2 Stored Procedure Builder online를 참조하십시오.

Stored Procedure Builder 사용 이점

Stored Procedure Builder는 저장 프로시듀어를 작성하고 설치하며 테스트할 때 사용하기 쉬운 개발 환경을 제공하여 DB2 서버에서 저장 프로시듀어 등록, 구축 및 설치 작업의 세부사항보다 저장 프로시듀어 논리 작성에 초점을 맞출 수 있게 도와 줍니다. Stored Procedure Builder는 저장 프로시듀어를 개발하는 플랫폼과 다른 서버 플랫폼에서 저장 프로시듀어를 구축할 수 있게 하여 상호 플랫폼 응용 프로그램 개발을 지원합니다.

새로운 저장 프로시듀어 작성

Stored Procedure Builder를 사용하면 DB2 데이터베이스 서버에서의 저장 프로시듀어 작성 및 설치 과정이 단순화됩니다. 저장 프로시듀어 마법사 및 SQL 지원은 저장 프로시듀어를 쉽게 개발할 수 있게 합니다.

Stored Procedure Builder에서 Java 또는 SQL로 작성된 매우 휴대하기 좋은 저장 프로시듀어를 작성할 수 있습니다. 저장 프로시듀어 마법사를 사용하여 기본 SQL 구조를 작성한 다음, 소스 코드 편집기를 사용하여 저장 프로시듀어를 수정함으로써 정교한 저장 프로시듀어 논리를 포함할 수 있습니다.

저장 프로시듀어 작성시 단일 결과 세트, 다중 결과 세트 또는 출력 매개변수만 리턴하도록 선택할 수 있습니다. 저장 프로시듀어가 데이터베이스 테이블을 작성하거나 갱신할 때 결과 세트를 리턴하지 않도록 선택할 수 있습니다. 저장 프로시듀어 마법사를 사용하여 저장 프로시듀어가 클라이언트 응용프로그램에서 호스트 변수의 값을 수신하도록 이에 대한 입출력 매개변수를 정의할 수 있습니다. 추가로 저장 프로시듀어에서 다중 SQL문을 작성하여 저장 프로시듀어에서 대소문자 값을 수신한 다음 수 많은 조회 중에서 하나를 선택합니다.

목표 데이터베이스에서 저장 프로시듀어를 구축하려면 저장 프로시듀어에 있는 **완료**를 누르기만 하면 됩니다. CREATE PROCEDURE문을 사용하여 DB2에서 저장 프로시듀어를 수동으로 등록하지 않아도 됩니다.

기존의 저장 프로시듀어 작업

데이터베이스 서버에서 저장 프로시듀어를 성공적으로 구축한 후 프로시듀어를 수정, 재구축, 수행 및 테스트할 준비가 되어 있습니다. 저장 프로시듀어를 수정함으

로써 정교한 저장 프로시듀어 논리를 포함시키도록 메소드를 코드에 추가할 수 있습니다. Stored Procedure Builder에서 저장 프로시듀어를 열면 소스 코드가 편집기에 표시됩니다. 편집기는 Java 또는 SQL로 작성된 저장 프로시듀어에 대해 언어를 구분합니다.

Stored Procedure Builder로부터 저장 프로시듀어를 수행하면 제대로 설치되도록 프로시듀어를 테스트할 수 있습니다. 저장 프로시듀어를 실행할 때 이는 설정되는 방법에 따라 입력된 테스트 입력 매개변수 값을 기반으로 결과 세트를 리턴할 수 있습니다. 사용자는 저장 프로시듀어가 DB2 데이터베이스 서버에 올바로 설치되었음을 알고 있으므로 저장 프로시듀어 테스트는 클라이언트 응용프로그램 작성을 쉽게 만들어 줍니다. 그런 다음 클라이언트 응용프로그램 작성 및 디버깅에 촛점을 맞출 수 있습니다.

Stored Procedure Builder의 프로젝트 창에서 저장 프로시듀어를 제거하거나 이를 다른 데이터베이스 연결에 복사할 수도 있습니다.

Stored Procedure Builder 프로젝트 작성

새로운 또는 기존의 Stored Procedure Builder 프로젝트를 열 때 프로젝트 창은 사용자가 연결된 DB2 데이터베이스에 상주하는 모든 저장 프로시듀어를 표시합니다. 저장 프로시듀어를 필터하여 그 이름 또는 스키마를 기반으로 한 프로시듀어를 볼 수 있습니다. Stored Procedure Builder 프로젝트는 데이터베이스에 제대로 구축되지 않은 연결 정보 및 저장 프로시듀어 오브젝트만 저장합니다.

저장 프로시듀어 디버깅

Stored Procedure Builder 및 IBM 분산 디버거(별도 사용 가능)를 사용하면, DB2 서버에 설치된 저장 프로시듀러를 원격으로 디버그할 수 있습니다. 저장 프로시듀어를 디버그하려면, 저장 프로시듀어를 디버그 모드에서 구축하고 클라이언트 IP 주소에 대해 항목을 디버그하며 저장 프로시듀어를 실행합니다. 응용프로그램 내부로부터 저장 프로시듀어를 디버그할 필요가 없습니다. 호출 응용프로그램을 테스트하는 것과 저장 프로시듀어를 테스트하는 것을 분리시킬 수 있습니다.

Stored Procedure Builder를 사용하면 저장 프로시듀어 디버그 테이블에서 디버그 항목을 변경, 추가 또는 제거할 수 있는 권한이 있는 모든 저장 프로시듀어를

볼 수 있습니다. 데이터베이스 관리자이거나 선택된 저장 프로시듀어의 작성자이면, 다른 사용자가 저장 프로시듀어를 디버그할 수 있는 권한을 부여할 수 있습니다.

제4부 오브젝트 관계형 프로그래밍

© Copyright IBM Corp. 1993, 2000

제10장 오브젝트 관계형 기능 사용

DB2 오브젝트 확장 사용 이유 297
DB2의 오브젝트 관계형 기능. 297
사용자 정의 구별 유형 300
오브젝트에 대해 동작 정의: 사용자 정의 루틴 301

DB2 오브젝트 확장 사용 이유

현재 프로그래밍 언어 기술에서의 가장 중요한 최신 개발 중의 하나는 *오브젝트 지향*입니다. 오브젝트 지향이란 응용프로그램 도메인의 엔터티가 분류를 통해 상호 관련된 독립적인 오브젝트들로서 모델링될 수 있는 개념을 말합니다. 오브젝트의 외부 동작과 특성은 객관화되는 반면 오브젝트의 내부 구현 세부 사항은 숨겨져 있습니다. 오브젝트 지향을 통해 사용자 응용프로그램 도메인 및 관련 유형들로 구성된 그룹에 있는 오브젝트들 사이의 유사점 및 차이점을 캡쳐할 수 있습니다. 같은 유형의 오브젝트는 같은 유형별 작동 세트를 공유하므로, 응용프로그램 도메인에서의 오브젝트 작동을 반영합니다.

DB2의 오브젝트 확장을 통해 관계형 기술의 강점을 구축하는 동시에 오브젝트 기술의 많은 장점을 실현시킬 수 있습니다. 관계형 시스템에서 데이터 유형은 이러한 데이터 유형의 인스턴스(또는 오브젝트)가 저장되어 있는 테이블 컬럼의 데이터를 설명합니다. 이 인스턴스에 대한 작업은, 표현식이 허용되는 곳에서 호출될 수 있는 함수 또는 연산자를 통해 지원됩니다.

DB2의 오브젝트 확장을 이용하여, 오브젝트 지향(OO) 개념 및 방법론을 사용자의 관계형 데이터베이스에 통합시킬 수 있습니다.

DB2의 오브젝트 관계형 기능

오브젝트 지향 방식으로 데이터를 모델화하는 데 도움을 주는 일부 오브젝트 관계형 기능에는 다음이 들어 있습니다.

대형 오브젝트에 대한 데이터 유형

텍스트, 오디오, 엔지니어링 데이터 또는 비디오와 같이 시스템에서 모델화해야 하는 데이터는 매우 크고 복잡할 수 있습니다. VARCHAR 또는 VARGRAPHIC 데이터 유형은 이 크기의 오브젝트에 대해 충분히 크지

© Copyright IBM Corp. 1993, 2000

않을 수도 있습니다. DB2는 데이터 오브젝트를 최고 2GB 크기의 문자여로 저장하는 데이터 유형을 3 가지 제공합니다. 세 가지 데이터 유형은, 2진 대형 오브젝트(BLOB), 1바이트 문자 대형 오브젝트(BLOB) 및 2바이트 문자 대형 오브젝트(CLOB)를 말합니다.

사용자 정의 데이터 유형

사용자가 정의한 유형을 사용하면 오브젝트의 의미를 제어할 수 있습니다. 예를 들어, 응용프로그램은 "text"나 "address"라고 하는 유형이 필요할 수 있습니다. 이런 유형은 내장 유형으로 존재하지 않습니다. 그러나 DB2의 오브젝트 관계형 기능을 사용하면 이런 유형을 정의하고 데이터베이스에서 사용할 수 있습니다.

사용자가 정의한 유형은 다음 방법으로 더 자세히 분류될 수 있습니다.

구별 유형

구별 유형은 기존 DB2 내장 데이터 유형을 기반으로 합니다. 즉, 내부적으로 이는 내장 유형과 같지만 사용자는 이런 유형의 의미를 정의할 수 있습니다. DB2에는 또한 매우 큰 오브젝트를 저장하고 처리하기 위한 내장 유형이 있습니다. 구별 유형은 오디오 또는 비디오 스트림과 같은 것에 사용할 수 있는 대형 오브젝트(LOB) 데이터 유형 중 하나를 기반으로 사용할 수 있습니다.

구조화된 유형

구조화된 유형은 오브젝트 속성을 단일 유형으로 모으는 방법입니다.

사용자가 정의한 동작

자체 루틴을 SQL이나 외부 언어로 작성하여 DB2가 자신의 오브젝트에 대해 작업하도록 만들 수 있습니다. 사용자가 정의한 루틴에는 두 가지가 있습니다.

사용자 정의 함수(UDFs)

UDF는 내장 함수 또는 연산자와 같이 SQL 조회에서 어떤 것이 오브젝의 처리를 지원하는지 정의할 수 있는 함수입니다. UDF를 사용하여 사용자가 정의한 유형뿐만 아니라 모든 유형의 컬럼 값을 처리할 수 있습니다.

사용자 정의 메소드

UDF 처럼 메소드는 오브젝트의 동작을 정의하지만 이는 사용자가 정의한 특정 구조화된 유형과 밀접하게 캡슐화됩니다.

색인 확장

색인 확장을 사용하면 DB2 색인 구조화된 유형과 구별 유형을 지정할 수 있습니다. 색인 확장을 작성하려면 CREATE INDEX EXTENSION문을 실행해야 합니다. CREATE INDEX EXTENSION문은 구조화된 유형이나 구별 유형의 값을 색인 키로 변환하고 DB2가 이런 색인 키를 검색하여 그 성능을 최저화하는 방법을 정의하는 외부 테이블 함수를 지정합니다.

테이블 함수 작성에 대한 자세한 내용은 433 페이지의 『제15장 사용자 정의 함수(UDF)』를 참조하십시오. 색인 확장을 사용하여 구조화된 유형과 구별 유형을 사용하여 응용프로그램의 성능을 개선하는 색인 확장 사용에 대한 자세한 내용은 *관리 안내서*을 참조하십시오. CREATE INDEX문에 대한 자세한 정보는 *SQL 참조서* 책을 참조하십시오.

제한조건

제한조건은 데이터베이스에서 강요하는 사용자가 정의한 규칙입니다. 4가지 유형의 제한조건이 있습니다.

고유성 키의 고유 값이 테이블에 있어야 합니다. 고유성을 위해 키를 구성하는 컬럼에 대한 모든 변경을 검사합니다.

참조 무결성

삽입, 갱신 및 삭제 조작에 참조 제한조건을 강요합니다. 이는 모든 외부 키의 모든 값이 유효한 데이터베이스 상태입니다.

테이블 점검

변경된 데이터가 테이블이 작성되거나 변경되었을 때 지정된 조건을 위반하지 않았는지 검증합니다.

트리거 트리거는 테이블에 연관되어 있고 그 테이블에서 데이터 변경 조작이 일어날 때 자동으로 활성화되는 SQL문으로 구성됩니다. 트리거를 사용하여 사용자는 비지니스 규칙과 같은 일반적인 형태의 무결성을 지원합니다.

고유 제한조건, 참조 무결성 그리고 테이블 점검 제한조건에 대한 자세한 내용은 *관리 안내서*를 참조하십시오. 트리거에 대해서는 537 페이지의 『제 16장 활동중인 DBMS에서 트리거 사용』의 내용을 참조하십시오.

전형적인 응용프로그램에서의 오브젝트 지향 기능 사용

DB2의 오브젝트 지향 기능 중에는 중요한 시너지 효과가 있습니다. DB2 오브젝트 지향 메카니즘의 사용은 오브젝트 지향 응용프로그램 지원에 제한되지 않습니다. 대중적인 오브젝트 지향 프로그래밍 언어인 C++를 사용하여 모든 종류의 오브젝트 지향이 아닌 응용프로그램을 구현하듯이, DB2가 제공하는 오브젝트 지향 메카니즘 역시 모든 종류의 오브젝트 지향이 아닌 응용프로그램을 지원하는 데 매우 유용합니다. DB2의 오브젝트 관련 기능은 모든 데이터베이스 응용프로그램 모델화에 사용될 수 있는 범용 메카니즘입니다. 이런 이유로 DB2 오브젝트 확장은 전형적인 응용프로그램에 대한 지원 개선 외에도, 전형적이지 않은 즉, 오브젝트 지향 응용프로그램에 대한 광범위한 지원을 제공합니다.

사용자 정의 구별 유형

구별 유형은 기존 내장 유형을 기반으로 합니다. 예를 들어, 미국 달러 및 캐나다 달러와 같은 다양한 통화를 나타내는 구별 유형이 있을 수 있습니다. 이런 유형은 둘다 내부적(그리고 호스트 언어 프로그램으로)으로 이런 통화를 정의하는 내장 유형으로 표현됩니다. 예를 들어, 양쪽 통화를 DECIMAL로 정의하면 이는 시스템에서 소수점 데이터 유형으로 표현됩니다.

명백한 유형 지정

비록 동일한 내장 유형을 기반으로 여러 구별 유형을 사용할 수 있지만 구별 유형에는 *명백한 유형 지정*이라는 특성이 있습니다. 이 명백한 유형 지정의 특성을 가지고 이런 유형의 인스턴스를 그 유형의 다른 인스턴스 이외의 것과 직접 비교할 수 없습니다. 이는 미국 달러와 캐나다 달러는 먼저 변환하지 않고 바로 더하는 것과 같이 의미상 문제가 있는 조작을 방지합니다. 구별 유형의 인스턴스에 대해 발생할 수 있는 조작 유형을 정의합니다.

작동 유형

미국 달러 또는 캐나다 달러의 인스턴스에서 허용되는 조작을 어떻게 정의합니까? 사용자가 정의한 함수를 사용하여 구별 유형의 인스턴스에 허

용되는 동작을 정의하십시오. 미국 달러를 입력으로 받는 내장된 더하기 조작과 같은 함수를 등록하여 미국 달러의 인스턴스를 더하는 것과 같이 간단한 작업을 수행할 수 있습니다. 이런 종류의 함수를 정의하기 위해 응용프로그램을 코딩하지 않아도 됩니다.

그러나, 미국 달러 유형을 입력을 받고 이를 캐나다 달러 유형으로 변환하는 좀더 복잡한 함수를 작성할 수도 있습니다. 사용자 정의 함수에 대해서는 409 페이지의 『제14장 사용자 정의 함수(UDFs)와 메소드』의 내용을 참조하십시오.

제한조건을 사용하여 무결성 규칙을 구현할 수 있습니다.

대형 오브젝트(LOB)

구별 유형으로 모델화할 수 있는 오브젝트는 매우 큽니다. DB2에는 또한 매우 큰 오브젝트를 저장하고 처리하기 위한 새 내장된 유형이 있습니다. 구별 유형은 오디오 또는 비디오와 같은 것에 사용할 수 있는 대형 오브젝트(LOB) 데이터 유형 중 하나를 기반으로 사용할 수 있습니다.

오브젝트에 대해 동작 정의: 사용자 정의 루틴

오브젝트의 동작을 정의하려면 사용자 정의 함수(UDF)와 메소드를 사용할 수 있습니다.

사용자 정의 함수

UDF는 내장 함수 또는 연산자와 같이 SQL 조회에서 어떤 것이 오브젝트의 처리를 지원하는지 정의할 수 있는 함수입니다.(UDF는 사용자 정의 유형뿐만 아니라 모든 유형의 컬럼 값을 처리할 때 사용될 수 있습니다.) 따라서, 인스턴스는 테이블 컬럼에 저장되고 SQL 조회에서 UDF에 의해 조작됩니다. 예를 들어, LENGTH 구별 유형의 인스턴스과 WIDTH 구별 유형의 인스턴스를 취하고 영역을 계산하며, 이를 조회에 리턴하는 AREA 함수를 정의할 수 있습니다.

```
SELECT ID, area(length, width) AS area
FROM Property
WHERE area > 10000;
```

메소드 UDF와 같이 메소드는 오브젝트의 동작을 정의하지만 다음과 같은 점에서 함수와 다릅니다.

- 메소드는 특정 사용자 정의 구조 유형과 밀접하게 연관되어 있고 사용자 정의 유형과 같은 스키마에 저장됩니다.
- 메소드는 컬럼에 값으로 저장되는 사용자 정의 구조화 유형에서 호출되거나 구조화된 유형의 범위 지정된 참조에서 참조 해제 연산자(->)를 사용하여 호출될 수 있습니다.
- 메소드는 함수를 호출할 때 사용되는 다른 SQL 구문을 사용하여 호출될 수 있습니다.
- DB2는 메소드가 호출된 유형에서 시작하여 메소드에 대해 규정되지 않은 참조를 분석합니다. 메소드가 호출된 유형이 메소드를 정의하지 않으면, DB2는 메소드가 호출된 유형의 상위 유형에서 메소드를 호출하여 메소드를 분석합니다.

컬럼에 저장된 구조화된 유형에서 메소드를 호출하려면 구조화된 유형(또는 구조화된 유형을 분석하는 표현식)의 이름과 메소드 호출 연산자(..), 메소드의 이름을 포함합니다. 구조화된 유형의 범위 지정된 참조에서 메소드를 호출하려면 참조 해제 연산자(->), 메소드 호출 연산자 그리고 메소드의 이름을 사용하여 구조화된 유형에 대한 참조를 포함합니다.

DB2의 오브젝트 관계형 기능에 대한 자세한 내용은 다음을 참조하십시오.

- 315 페이지의『제12장 복잡한 오브젝트 사용: 사용자 정의 구조화 유형』
- 303 페이지의『제11장 사용자 정의 구별 유형』
- 383 페이지의『제13장 대형 오브젝트(LOB) 사용』
- 409 페이지의『제14장 사용자 정의 함수(UDFs)와 메소드』
- 433 페이지의『제15장 사용자 정의 함수(UDF)』
- 537 페이지의『제16장 활동중인 DBMS에서 트리거 사용』

제11장 사용자 정의 구별 유형

구별 유형을 사용하는 이유 303
구별 유형 정의 304
규정되지 않은 구별 유형 분석 305
CREATE DISTINCT TYPE 사용 예. . . 305
예: 비용 305
예: 작업 응용프로그램 306
구별 유형에서 테이블 정의. 306
예: 판매 306
예: 응용프로그램 양식 307
구별 유형 조작 307
구별 유형 조작의 예. 308
예: 상수와 구별 유형간의 비교 308
예: 구별 유형간 유형변환 309
예: 구별 유형에 관련된 포함 311
예: 구별 유형과 관련된 전래 UDF . . . 311
예: 구별 유형과 관련된 지정 312
예: 동적 SQL에서의 지정 312
예: 서로 다른 구별 유형이 관련된 지정 313
예: UNION에서의 구별 유형 사용 . . . 314

구별 유형을 사용하는 이유

사용자의 DB2 응용프로그램에서 작성한 *사용자 정의 구별 유형(UDT)*이라는 데이터 유형을 사용할 수 있습니다. 구별 유형에 연관된 몇 가지 이점은 다음과 같습니다.

1. **확장성**

   새로운 유형을 정의함으로써 응용프로그램을 지원하기 위해 DB2가 제공한 유형 세트를 증가시킬 수 있습니다.

2. **융통성**

   시스템에서 사용 가능한 유형의 다양성을 증대시키기 위해 사용자 정의 함수(UDF)를 사용하여 새로운 유형에 대한 의미 및 작동을 지정할 수 있습니다. UDF에 대한 세부사항은 409 페이지의 『제14장 사용자 정의 함수(UDFs)와 메소드』의 내용을 참조하십시오.

3. **일관성 있는 조작**

   명백한 유형 지정은 구별 유형이 제대로 작동하도록 합니다. 구별 유형에 정의된 함수만 구별 유형의 인스턴스에 적용될 수 있습니다.

4. **캡슐화**

구별 유형에 적용시킬 수 있는 연산자 및 함수 세트는 사용자 UDT의 작동을 정의합니다. 수행 응용프로그램은 사용자 유형에 대해 선택하는 내부 표시에 의존하지 않으므로, 구현에 있어서 융통성이 제공됩니다.

5. **성능**

   구별 유형은 데이터베이스 관리 프로그램으로 통합됩니다. 구별 유형이 내장 데이터 유형과 같은 방식으로 내부적으로 표시되므로, 내장 데이터 유형에 대해 내장 함수, 비교 연산자, 색인 등을 구현할 때 사용되는 동일한 유효 코드를 공유합니다.

구별 유형 정의

테이블, 색인 및 UDF와 같은 기타 오브젝트와 마찬가지로, 구별 유형은 CREATE 문으로 정의해야 합니다.

CREATE DISTINCT TYPE문을 사용하여 새로운 구별 유형을 정의하십시오. 명령문 구문 및 모든 옵션에 대한 상세한 설명은 *SQL 참조서*의 내용을 참조하십시오.

CREATE DISTINCT TYPE문의 경우 다음 사항에 유의하십시오.

1. 새로운 구별 유형 이름은 규정화되거나 규정화되지 않은 이름이 될 수 있습니다. 명령문의 권한 부여 ID와 다른 스키마에 의해 규정화되는 경우, 데이터베이스에 대해 DBADM 권한이 있어야 합니다.
2. 구별 유형의 소스 유형은 구별을 내부적으로 표시하기 위해 구별 유형에 의해 사용되는 유형입니다. 이러한 이유로 이는 내장 데이터 유형이어야 합니다. 이전에 정의된 구별 유형은 다른 구별 유형의 소스 유형으로서 사용될 수 없습니다.
3. WITH COMPARISONS절은 구별 유형 인스턴스에 대한 비교 조작이 DB2에 의해 생성되도록 이 함수가 지정하도록 하는 데 사용됩니다. 이 절은 소스 유형(예를 들어, INTEGER 및 DATE)에서 비교 조작이 지원될 경우 필수이고, 비교 조작이 지원되지 않을 경우(예를 들어, LONG VARCHAR 및 BLOB) 금지됩니다.

주: 구별 유형 정의의 일부로서 DB2는 항상 유형변환(cast) 함수를 생성합니다.

- 소스 유형의 표준 이름을 사용하여 구별 유형으로부터 소스 유형으로의 유형변환. 예를 들어, FLOAT에 기초하여 구별 유형을 작성할 경우 DOUBLE이라는 유형변환(cast) 함수가 작성됩니다.
- 소스 유형으로부터 구별 유형으로의 유형변환. 구별 유형으로의 추가 유형변환 생성 시기에 대해서는 *SQL 참조서*의 내용을 참조하십시오.

이 함수는 조회 중 구별 유형 조작에 대해 중요합니다.

규정되지 않은 구별 유형 분석

유형 이름 또는 함수가 다음과 같은 경우를 제외하고 함수 경로는 규정화되지 않은 유형 이름 또는 함수에 대한 참조를 분석합니다.

- 작성
- 제거
- 주석

규정되지 않은 함수 참조의 분석 방법에 대해서는 426 페이지의 『규정화된 함수 참조 사용』의 내용을 참조하십시오.

CREATE DISTINCT TYPE 사용 예

다음은 CREATE DISTINCT TYPE 사용 예입니다.

예: 비용

다른 통화를 처리할 수 있는 응용프로그램을 작성하고 있고, DB2가 조회에 있어서 서로 직접 이러한 통화를 비교 또는 조작할 수 없도록 하려 한다고 가정하십시오. 다른 통화값을 서로 비교하려고 할 때마다 변환이 필요합니다. 따라서 필요한 만큼의 구별 유형, 즉 표시해야 하는 각 통화에 대해 구별 유형 하나를 정의합니다.

```
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL (9,2) WITH COMPARISONS
CREATE DISTINCT TYPE CANADIAN_DOLLAR AS DECIMAL (9,2) WITH COMPARISONS
CREATE DISTINCT TYPE EURO AS DECIMAL (9,2) WITH COMPARISONS
```

DECIMAL(9, 2)에서 비교 연산자가 지원되므로 WITH COMPARISONS을 지정해야 합니다.

예: 작업 응용프로그램

사용자 회사의 구직자가 작성한 양식을 DB2 테이블에 기록하고, 이 양식의 정보를 추출하는 기능을 사용하려 한다고 가정하십시오. 이 기능은 (리턴할 정보를 찾을 수 없어) 일반 문자열에는 적용할 수 없으므로 작성된 양식을 표시하려면 구별 유형을 정의해야 합니다.

```
CREATE DISTINCT TYPE PERSONAL.APPLICATION_FORM AS CLOB(32K)
```

DB2가 CLOB에서의 비교를 지원하지 않으므로 WITH COMPARISONS절을 지정하지 않습니다. DBADM 권한이 있으므로 사용자의 권한 부여 ID와 다른 스키마 이름을 지정했고, 구직자 양식을 취급하는 모든 구별 유형 및 UDF를 같은 스키마로 기록하려 합니다.

구별 유형에서 테이블 정의

여러 개의 구별 유형을 정의한 후, 유형이 구별 유형인 컬럼으로 테이블을 정의할 수 있습니다. 다음은 CREATE TABLE 사용 예입니다.

예: 판매

다른 나라에서의 사용자 회사 판매 현황을 기록하기 위해 다음과 같이 테이블을 정의한다고 가정하십시오.

```
CREATE TABLE US_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR          INTEGER CHECK (YEAR > 1985),
   TOTAL         US_DOLLAR)

CREATE TABLE CANADIAN_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR          INTEGER CHECK (YEAR > 1985),
   TOTAL         CANADIAN_DOLLAR)

CREATE TABLE GERMAN_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR          INTEGER CHECK (YEAR > 1985),
   TOTAL         EURO)
```

위 예에서 구별 유형은 305 페이지의 『예: 비용』에서 동일한 CREATE DISTINCT TYPE문을 사용하여 작성됩니다. 이 예에서는 점검 제한조건을 사용합니다. 점검 제한조건에 대한 세부사항은 *SQL 참조서*의 내용을 참조하십시오.

예: 응용프로그램 양식

구직자가 작성한 양식을 기록하는 테이블을 다음과 같이 정의해야 한다고 가정하십시오.

```
CREATE TABLE APPLICATIONS
  (ID                 SYSIBM.INTEGER,
   NAME               VARCHAR (30),
   APPLICATION_DATE   SYSIBM.DATE,
   FORM               PERSONAL.APPLICATION_FORM)
```

규정자가 사용자의 권한 부여 ID와 같지 않고 기본 함수 경로를 변경하지 않았으므로 구별 유형 이름을 완전히 규정했습니다. 유형 및 함수 이름이 완전 규정되지 않을 때마다 DB2는 현재 함수 경로에 나열된 스키마를 검색하고, 규정되지 않은 현재 이름과 일치하는 함수 이름 또는 유형을 검색합니다. 현재 함수 경로에서 (생략된 경우) SYSIBM이 항상 고려되므로 내장 데이터 유형의 규정을 생략할 수 있습니다. 예를 들어, SET CURRENT FUNCTION PATH = cheryl을 실행할 수 있고, 현재 함수 경로 특수 레지스터 값은 "CHERYL"이 되고, "SYSIBM"이 포함되지 않습니다. 이제, CHERYL.INTEGER 유형이 정의되지 않았다면, CREATE TABLE FOO(COL1 INTEGER)문은 SYSIBM이 항상 SYSIBM.INTEGER 유형의 COL1로서 고려되므로 여전히 성공합니다.

그러나, 원할 경우 내장 데이터 유형을 완전히 규정할 수 있습니다. 현재 함수 경로 사용에 대한 세부사항은 *SQL 참조서*에 있습니다.

구별 유형 조작

구별 유형과 연관된 가장 중요한 개념 중 하나는 *명백한 유형 지정*입니다. 명백한 유형 지정은 구별 유형에 정의된 연산자 및 함수만 인스턴스에 적용될 수 있도록 합니다.

구별 유형의 인스턴스가 올바로 되도록 하려면 명백한 유형 지정이 중요합니다. 예를 들어, 현재 환율에 따라 미화 달러를 캐나다 달러로 변환시키는 함수가 정의된 경우 틀린 값을 리턴하므로, 유로화를 캐나다 달러로 변환시킬 때 같은 함수를 사용하지 않을 것입니다.

명백한 유형 지정 결과, DB2는 사용자가 구별 유형 인스턴스와 구별 유형에 대해 소스 유형의 인스턴스를 비교하는 조회를 작성하는 것을 허용하지 않습니다. 같은 이유로 DB2는 다른 유형에 정의된 함수를 구별 유형에 적용하는 것을 허용하지 않습니다. 다른 유형의 인스턴스와 구별 유형 인스턴스를 비교하려면 어떤 유형의 인스턴스를 유형변환해야 합니다. 같은 의미에서, 이 함수를 구별 유형 인스턴스에 적용하려면, 구별 유형 인스턴스를 구별 유형에 정의되지 않은 함수의 매개변수 유형으로 유형변환해야 합니다.

구별 유형 조작의 예

다음은 구별 유형 조작의 예입니다.

- 예: 상수와 구별 유형간의 비교
- 예: 구별 유형간 유형변환
- 예: 구별 유형에 관련된 포함
- 예: 구별 유형과 관련된 전래 UDF
- 예: 구별 유형과 관련된 지정
- 예: 동적 SQL에서의 지정
- 예: 서로 다른 구별 유형이 관련된 지정
- 예: UNION에서의 구별 유형 사용

예: 상수와 구별 유형간의 비교

1999년 7월(7/99) 미화 $100 000.00 이상으로 판매된 제품을 알려고 한다고 가정하십시오.

```
SELECT PRODUCT_ITEM
  FROM   US_SALES
  WHERE  TOTAL > US_DOLLAR (100000)
  AND    month = 7
  AND    year  = 1999
```

DECIMAL과 같은 미화 달러의 소스 유형 인스턴스와 미화 달러를 직접 비교할 수 없으므로 DB2가 제공하는 유형변환(cast) 함수를 사용하여 DECIMAL에서 미화 달러로 유형변환했습니다. DB2가 제공하는 다른 유형변환(cast) 함수(즉, 미화 달러에서 DECIMAL로 유형변환시키는 함수)를 사용할 수도 있고 모든 컬럼을 DECIMAL로 유형변환할 수 있습니다. 구별 유형으로 유형변환하거나 구별 유형을 유형변환할 경우에도, 유형변환 스펙 표기법을 사용하여 유형변환 또는 함수 표기를 수행할 수 있습니다. 즉, 조회를 다음과 같이 작성할 수 있습니다.

```
SELECT PRODUCT_ITEM
  FROM   US_SALES
  WHERE  TOTAL > CAST (100000 AS us_dollar)
  AND    MONTH = 7
  AND    YEAR  = 1999
```

예: 구별 유형간 유형변환

캐나다 달러를 미화 달러로 변환시키는 UDF를 정의하려 한다고 가정하십시오. DB2 외부에서 관리되는 파일에서 현재 환율을 알아볼 수 있습니다. 그러면 캐나다 달러로 값을 표시하여 환율 파일에 액세스하고, 해당 값을 미화 달러로 리턴하는 UDF를 정의하게 됩니다.

처음에는 이러한 UDF가 작성하기 쉬워 보일 것입니다. 그러나, C는 DECIMAL 값을 지원하지 않습니다. 다른 통화를 나타내는 구별 유형이 DECIMAL로서 정의됩니다. 사용자의 UDF는 십진수 정밀도를 유지하고 DECIMAL 값 표시가 허용되는 C에 의해 제공된 유일한 데이터 유형이므로, DOUBLE 값을 받아 리턴해야 합니다. 따라서, 사용자의 UDF는 다음과 같이 정의되어야 합니다.

```
CREATE FUNCTION CDN_TO_US_DOUBLE(DOUBLE) RETURNS DOUBLE
  EXTERNAL NAME '/u/finance/funcdir/currencies!cdn2us'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED
```

캐나다 달러와 미달러 사이의 환율은 두 번의 UDF 호출 사이에 변경될 수 있으므로, NOT DETERMINISTIC으로 선언합니다.

당면 문제는 어떻게 캐나다 달러를 이 UDF로 전달하고 이 UDF에서 미화 달러를 가져올 것인지입니다. 캐나다 달러는 DECIMAL 값으로 유형변환되어야 합니다. DECIMAL 값은 DOUBLE로 유형변환되어야 합니다. 리턴된 DOUBLE 값을 DECIMAL로, DECIMAL 값을 미화 달러로 유형변환하도록 해야 합니다.

이러한 유형변환은 전래 UDF를 정의할 때마다 DB2에 의해 자동으로 수행되고, 전래 UDF 매개변수 및 리턴 유형이 매개변수와 정확하게 일치하지 않으며 소스 함수 유형을 리턴합니다. 따라서, 두 개의 전래 UDF를 정의해야 합니다. 첫번째 유형은 DOUBLE 값을 DECIMAL 표시로 가져옵니다. 두 번째 유형은 DECIMAL 값을 구별 유형으로 가져옵니다. 즉, 다음과 같이 정의합니다.

```
CREATE FUNCTION CDN_TO_US_DEC (DECIMAL(9,2)) RETURNS DECIMAL(9,2)
  SOURCE CDN_TO_US_DOUBLE (DOUBLE)

CREATE FUNCTION US_DOLLAR (CANADIAN_DOLLAR) RETURNS US_DOLLAR
  SOURCE CDN_TO_US_DEC (DECIMAL())
```

US_DOLLAR(C1)에서처럼 US_DOLLAR 함수의 호출이 다음을 호출하는 것과 같은 효과입니다. 여기서 C1은 유형이 캐나다 달러인 컬럼입니다.

```
US_DOLLAR (DECIMAL(CDN_TO_US_DOUBLE (DOUBLE (DECIMAL (C1)))))
```

즉, C1(캐나다 달라)은 10진수로 유형변환되고, 다시 2배 값으로 유형변화되어 CDN_TO_US_DOUBLE 함수로 전달됩니다. 이 함수는 환율 파일에 액세스하여 10진수로, 다시 미화 달러로 유형변환되는 2배 값(미화 달러의 양 표시)을 리턴합니다.

유로화를 미화 달러로 변환하는 함수는 위 예와 유사합니다.

```
CREATE FUNCTION EURO_TO_US_DOUBL(DOUBLE)
  RETURNS DOUBLE
  EXTERNAL NAME '/u/finance/funcdir/currencies!euro2us'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED

CREATE FUNCTION EURO_TO_US_DEC (DECIMAL(9,2))
  RETURNS DECIMAL(9,2)
  SOURCE EURO_TO_US_DOUBL (DOUBLE)
```

```
CREATE FUNCTION US_DOLLAR(EURO) RETURNS US_DOLLAR
  SOURCE EURO_TO_US_DEC (DECIMAL())
```

예: 구별 유형에 관련된 포함

1999년 7월(7/1999) 한달 동안 캐나다 및 독일에서보다 미국에서 어떤 제품이 더 많이 팔렸는지 알아보려면 다음과 같이 하십시오.

```
SELECT US.PRODUCT_ITEM, US.TOTAL
  FROM US_SALES AS US, CANADIAN_SALES AS CDN, GERMAN_SALES AS GERMAN
  WHERE US.PRODUCT_ITEM = CDN.PRODUCT_ITEM
  AND US.PRODUCT_ITEM = GERMAN.PRODUCT_ITEM
  AND US.TOTAL > US_DOLLAR (CDN.TOTAL)
  AND US.TOTAL > US_DOLLAR (GERMAN.TOTAL)
  AND US.MONTH = 7
  AND US.YEAR  = 1999
  AND CDN.MONTH = 7
  AND CDN.YEAR  = 1999
  AND GERMAN.MONTH = 7
  AND GERMAN.YEAR  = 1999
```

미화 달러를 캐나다 달러나 유로화와 직접 비교하므로 UDF를 사용하여 캐나다 달러로 된 수량을 미화 달러로 유형변환하고, UDF를 사용하여 수량을 유로화에서 미화 달러로 유형변환합니다. 수량이 금액면에서 비교가능하지 않으므로, 이들 모두를 DECIMAL로 유형변환하고 변환된 DECIMAL 값을 비교할 수 없습니다. 즉, 금액은 동일한 통화로 이루어져 있지 않습니다.

예: 구별 유형과 관련된 전래 UDF

유로화에 대해 SUM을 지원하도록 내장 SUM 함수에 전래 UDF를 정의했다고 가정하십시오.

```
CREATE FUNCTION SUM (EUROS)
  RETURNS EUROS
  SOURCE SYSIBM.SUM (DECIMAL())
```

1994년 각 제품에 대해 독일에서의 총 판매액을 알고자 합니다. 미화 달러로 총 판매액을 알아볼 수 있습니다.

```
SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
  FROM GERMAN_SALES
  WHERE YEAR = 1994
  GROUP BY PRODUCT_ITEM
```

위와 유사한 방식으로 미화 달러로 SUM 함수를 정의하지 않은 한 SUM (us_dollar (total))을 작성할 수 없습니다.

예: 구별 유형과 관련된 지정

새로운 구직자가 작성한 양식을 데이터베이스에 저장한다고 가정하십시오. 작성된 양식을 표시하는 데 사용되는 문자열이 포함된 호스트 변수를 정의합니다.

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL TYPE IS CLOB(32K) hv_form;
EXEC SQL END DECLARE SECTION;

/* Code to fill hv_form */

INSERT INTO APPLICATIONS
  VALUES (134523, 'Peter Holland', CURRENT DATE, :hv_form)
```

DB2에서는 구별 유형의 소스 유형 인스턴스를 그 구별 유형을 사용하는 목표에 지정할 수 있으므로, 유형변환 함수를 명시적으로 호출하여 문자열을 구별 유형 personal.application_form으로 변환하지 마십시오.

예: 동적 SQL에서의 지정

동적 SQL에서 『예: 구별 유형과 관련된 지정』에 제공된 것과 같은 명령문을 사용하려면 다음과 같이 매개변수 표시문자를 사용할 수 있습니다.

```
EXEC SQL BEGIN DECLARE SECTION;
  long id;
  char name[30];
  SQL TYPE IS CLOB(32K) form;
  char command[80];
EXEC SQL END DECLARE SECTION;

/* Code to fill host variables */

strcpy(command,"INSERT INTO APPLICATIONS VALUES");
strcat(command,"(?, ?, CURRENT DATE, CAST (? AS CLOB(32K)))");

EXEC SQL PREPARE APP_INSERT FROM :command;
EXEC SQL EXECUTE APP_INSERT USING :id, :name, :form;
```

매개변수 표시문자 유형이 구별 유형 컬럼에 지정할 수 있는 유형인 CLOB(32K)라는 것을 DB2에게 알리기 위해 DB2의 유형변환 스펙을 사용했습니다. 호스트

언어가 구별 유형을 지원하지 않으므로, 구별 유형의 호스트 변수를 선언할 수 없습니다. 따라서, 매개변수 표시문자 유형을 구별 유형으로 지정할 수 없습니다.

예: 서로 다른 구별 유형이 관련된 지정

311 페이지의 『예: 구별 유형과 관련된 전래 UDF』에서 유로화에서 전래한 UDF와 유사하게, 미화 달러 및 캐나다 달러로 SUM을 지원하는 내장 SUM 함수에 두 개의 전래 UDF를 정의한다고 가정하십시오.

```
CREATE FUNCTION SUM (CANADIAN_DOLLAR)
  RETURNS CANADIAN_DOLLAR
  SOURCE SYSIBM.SUM (DECIMAL())

CREATE FUNCTION SUM (US_DOLLAR)
  RETURNS US_DOLLAR
  SOURCE SYSIBM.SUM (DECIMAL())
```

이제 감독자가 각 제품의 각 국가에서의 연간 총 판매액을 미화 달러로 별도의 테이블에서 유지보수하도록 요청한다고 가정하십시오.

```
CREATE TABLE US_SALES_94
  (PRODUCT_ITEM  INTEGER,
   TOTAL         US_DOLLAR)

CREATE TABLE GERMAN_SALES_94
  (PRODUCT_ITEM  INTEGER,
   TOTAL         US_DOLLAR)

CREATE TABLE CANADIAN_SALES_94
  (PRODUCT_ITEM  INTEGER,
   TOTAL         US_DOLLAR)

INSERT INTO US_SALES_94
  SELECT PRODUCT_ITEM, SUM (TOTAL)
  FROM US_SALES
  WHERE YEAR = 1994
  GROUP BY PRODUCT_ITEM

INSERT INTO GERMAN_SALES_94
  SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
  FROM GERMAN_SALES
  WHERE YEAR = 1994
  GROUP BY PRODUCT_ITEM
```

```
INSERT INTO CANADIAN_SALES_94
  SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
  FROM CANADIAN_SALES
  WHERE YEAR = 1994
  GROUP BY PRODUCT_ITEM
```

여러 구별 유형은 서로 직접 지정될 수 없으므로 캐나다 달러와 유로화로 된 금액을 미국 달러로 유형변환합니다. 구별 유형은 자체 소스 유형으로만 유형변환될 수 있으므로 유형변환 스펙 구문을 사용하지 마십시오.

예: UNION에서의 구별 유형 사용

회사의 모든 제품에 대한 총 판매액을 포함하여 미국인 사용자에게 뷰를 제공하고자 한다고 가정하십시오.

```
CREATE VIEW ALL_SALES AS
  SELECT PRODUCT_ITEM, MONTH, YEAR, TOTAL
  FROM US_SALES
  UNION
  SELECT PRODUCT_ITEM, MONTH, YEAR, US_DOLLAR (TOTAL)
  FROM CANADIAN_SALES
  UNION
  SELECT PRODUCT_ITEM, MONTH, YEAR, US_DOLLAR (TOTAL)
  FROM GERMAN_SALES
```

구별 유형은 같은 구별 유형과만 호환되므로, 캐나다 달러를 미국 달러로 유형변환하고 유로화를 미국 달러로 유형변환합니다. 유형변환 스펙은 구별 유형과 그 소스 유형간에만 유형변환할 수 있으므로, 구별 유형간에 유형변환할 때 함수 표기법을 사용해야 합니다.

제12장 복잡한 오브젝트 사용: 사용자 정의 구조화 유형

구조화 유형 개요 316
구조화 유형 계층 작성 317
참조 유형 및 그 표현 유형 319
참조 유형 유형변환 및 비교 320
기타 시스템-생성 루틴 321
유형 동작 정의 322
유형화 테이블에서의 오브젝트 저장 . . . 324
유형화 테이블에서의 오브젝트간 관계 정의 326
컬럼에서의 오브젝트 저장 327
구조화 유형의 추가 특성 328
유형화 테이블에서의 구조화 유형 사용. . . 329
유형화 테이블 작성 329
테이블의 유형 정의 330
오브젝트 식별자 명명 330
테이블 계층에서 위치 지정. 330
SELECT 특권 계승 지정 331
컬럼 옵션 정의 332
참조된 컬럼의 범위 정의 332
유형화 테이블에서 데이터 처리 중 . . . 333
참조 유형 사용 334
참조 유형 비교 335
참조를 사용한 의미 관계 정의 335
참조 무결성 및 범위 지정된 참조의 차이. 337
뷰 유형 작성 338
사용자 정의 유형(UDT) 또는 유형 맵핑 제거 340
뷰 변경 또는 제거 341
유형화 테이블 조회 342
참조를 해제하는 조회 342
DEREF 내장 함수 344
기타 유형-관련 내장 함수 344
추가 조회 스펙 기술. 345
ONLY를 사용하여 특정 유형의 오브젝트 리턴 345
TYPE 술어를 사용하여 리턴된 유형 제한 345
OUTER를 사용하여 가능한 모든 속성 리턴 346
도움말 및 추가 정보. 347
시스템 생성 오브젝트 식별자 정의 . . 347
오브젝트 식별자 컬럼에 대한 제한조건 작성 348
구조화 유형을 컬럼 유형으로 작성 및 사용 349
컬럼에 구조화 유형 인스턴스 삽입 . . . 349
구조화 유형 컬럼으로 테이블 정의 . . . 350
구조화 유형 속성으로 유형 정의. . . . 351
구조화 유형 값이 있는 행 삽입 351
구조화 유형 값의 검색 및 수정 353
검색 속성 353
부속 유형의 속성 액세스 354
속성 수정 355
유형에 대한 정보 리턴 355
유형과 변환 연관 355
변환 그룹 명명에 대한 권장사항. . . 356
변환 그룹 지정 위치. 358
외부 루틴에 대해 변환 그룹 지정 . . 358
동적 SQL에 대한 변환 그룹 설정 . . 359
정적 SQL에 대한 변환 그룹 설정 . . 359
호스트 언어 프로그램에 대한 맵핑 작성: 변환 함수 359
외부 루틴과의 오브젝트 변경: 함수 변환 360
변환 함수 요약 371
DB2에서 부속 유형 데이터 검색(바인드아웃) 372

© Copyright IBM Corp. 1993, 2000

DB2로 부속 유형 데이터 리턴(바인드 인) 377
구조화 유형 호스트 변수 작업 381
구조화 유형 호스트 변수 선언 . . . 381
구조화 유형 설명 382

구조화 유형 개요

구조화 유형은 속성으로 잘 정의된 구조가 들어 있는 오브젝트를 모델화할 때 유용합니다. 속성은 유형의 인스턴스를 설명하는 특성입니다. 예를 들어, 기하학적 모양은 데카르트식 좌표 목록을 그 속성으로 가지고 있을 수 있습니다. 사람은 이름, 주소 등의 속성이 있습니다. 부서에는 이름 또는 다른 종류의 ID가 있습니다.

유형을 작성하려면 유형의 이름, 그 속성 이름 그리고 데이터 유형을 지정하고, 선택적으로 이 유형의 참조 유형을 시스템에서 표현하는 방법을 지정해야 합니다. 여기에 BusinessUnit_t 유형을 작성하는 SQL이 있습니다.

```
CREATE TYPE BusinessUnit_t AS
  (Name VARCHAR(20),
  Headcount INT)
  REF USING INT
  MODE DB2SQL;
```

AS절은 유형에 연관된 속성 정의를 제공합니다. BusinessUnit_t는 Name과 Headcount의 두 속성이 있는 유형입니다. 구조화 유형을 작성하려면 CREATE TYPE문에 MODE DB2SQL절을 포함시켜야 합니다. REF USING절 사용에 대한 자세한 내용은 319 페이지의 『참조 유형 및 그 표현 유형』을 참조하십시오.

구조화 유형은 전형적인 관계 데이터 유형을 능가하는 두 가지 주요 확장을 제공합니다. 계승이 특성과 구조화 유형의 인스턴스를 테이블의 행으로 또는 컬럼의 값으로 저장하는 기능이 그것입니다. 다음 절에서 이런 기능을 간단하게 설명합니다.

계승 전형적인 관계 테이블과 컬럼을 사용하는 사람과 같은 오브젝트를 모델화할 수 있습니다. 그러나, 구조화 유형은 *계승*이라는 추가 특성을 제공합니다. 즉, 구조화 유형에는 그 속성을 전부 다시 사용하고 부속 유형에 고유한 추가 속성이 들어 있는 *부속 유형*이 있을 수 있습니다. 예를 들어, 구조화 유형 Person_t에는 Name, Age 및 Address에 대한 속성이 있습니다. Person_t의 부속 유형은 Name, Age, Address의 모든 속성과

SerialNum, Salary, BusinessUnit의 추가 속성이 들어 있는 Employee_t 입니다.

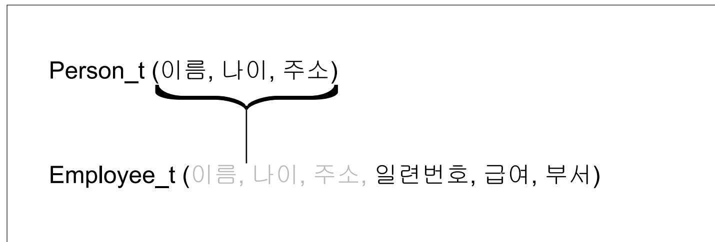


그림 7. 구조화 유형 Employee_t는 Person_t에서 계승합니다.

구조화 유형의 인스턴스 저장

구조화 유형 인스턴스는 다음과 같은 두 가지 방법으로 데이터베이스에 저장될 수 있습니다.

- 테이블의 각 컬럼이 유형 인스턴스의 속성인 테이블에 있는 행으로. 오브젝트를 테이블에 행으로 저장하려면 테이블 정의에서 개별 컬럼을 지정하는 대신 테이블은 구조화 유형으로 정의됩니다.

  ```
  CREATE TABLE Person OF Person_t
     ...
  ```

  테이블의 각 컬럼은 그 이름과 데이터 유형을 표시된 구조화 유형의 속성 중 하나에서 파생됩니다. 이런 테이블을 유형화 테이블이라고 합니다.

- 컬럼의 값으로. 오브젝트를 테이블 컬럼에 저장하려면 컬럼은 구조화 유형을 그 유형으로 사용하여 정의됩니다. 다음 명령문은 Address_t 구조화 유형의 Address 구조화 유형이 있는 Properties 테이블을 작성합니다.

  ```
  CREATE TABLE Properties
     (ParcelNum INT,
     Photo BLOB(2K),
     Address Address_t)
     ...
  ```

구조화 유형 계층 작성

구조화 유형은 새로 작성된 유형이 원래 구조 유형의 *부속 유형*인 다른 구조화 유형*하에서* 작성될 수 있습니다. 원래 유형은 *상위 유형*입니다. 부속 유형은 상위 유형의 모든 속성을 계승하고 선택적으로 자체 추가 속성을 가질 수도 있습니다.

예를 들어, 데이터 모델은 관리자라는 직원의 특수 유형을 나타내야 합니다. 관리자에게는 관리자가 아닌 직원보다 더 많은 속성이 있습니다. Manager_t 유형은 직원에 대해 정의된 속성을 계승하지만, 이는 관리자에게만 적용할 수 있는 특별 보너스와 같은 자체 추가 속성과 함께 정의되기도 합니다. 이 책의 예에서 사용되는 유형 계층은 그림8에 있습니다. Address_t의 유형 계층은 349 페이지의 『컬럼에 구조화 유형 인스턴스 삽입』에 정의되어 있습니다.

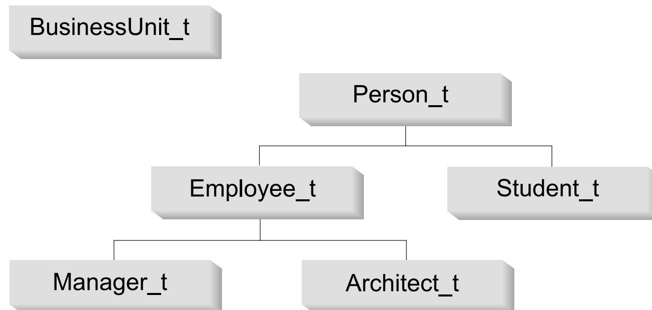


그림 8. 유형 계층 (BusinessUnit_t 및 Person_t)

그림8에서 사람 유형 Person_t는 계층의 루트 유형입니다. Person_t는 또한 그 밑에 있는 유형의 상위 유형인데, 이 경우 Employee_t로 명명된 유형과 Student_t로 명명된 유형입니다. 부속 유형과 상위 유형간의 관계는 변합니다. 다시 말해서 부속 유형과 상위 유형간의 관계는 전체 유형 계층 전반에 걸쳐 존재합니다. 그러므로 Person_t는 Manager_t와 Architect_t 유형의 상위 유형이기도 합니다.

316 페이지의 『구조화 유형 개요』에서 정의된 유형 BusinessUnit_t에는 부속 유형이 없습니다. 349 페이지의 『컬럼에 구조화 유형 인스턴스 삽입』에서 정의된 유형 Address_t에는 Germany_addr_t, Brazil_addr_t, US_addr_t의 부속 유형이 있습니다.

유형 Person_t에 대한 CREATE TYPE문은 Person_t를 INSTANTIABLE로 선언합니다. INSTANTIABLE 또는 NOT INSTANTIABLEF절을 사용하여 구조화 유형을 선언하는 것에 대한 자세한 내용은 328 페이지의 『구조화 유형의 추가 특성』을 참조하십시오.

다음 SQL문은 Person_t 유형 계층을 작성합니다.

```
CREATE TYPE Person_t AS
   (Name VARCHAR(20),
   Age INT,
   Address Address_t)
```

```
      INSTANTIABLE
      REF USING VARCHAR(13) FOR BIT DATA
      MODE DB2SQL;

   CREATE TYPE Employee_t UNDER Person_t AS
      (SerialNum INT,
      Salary DECIMAL (9,2),
      Dept REF(BusinessUnit_t))
      MODE DB2SQL;

   CREATE TYPE Student_t UNDER Person_t AS
      (SerialNum CHAR(6),
      GPA DOUBLE)
      MODE DB2SQL;

   CREATE TYPE Manager_t UNDER Employee_t AS
      (Bonus DECIMAL (7,2))
      MODE DB2SQL;

   CREATE TYPE Architect_t UNDER Employee_t AS
      (StockOption INTEGER)
      MODE DB2SQL;
```

Person_t에는 Name, Age, Address의 세 가지 속성이 있습니다. 이 속성의 부속 유형인 Employee_t와 Student_t는 각각 Person_t의 속성을 계승하고, 특정 유형에 고유한 여러 개의 추가 속성이 있습니다. 예를 들어, 직원 및 학생에게 일련 번호가 있더라도, 학생 일련 번호에 사용되는 형식은 직원 일련 번호에 사용되는 형식과 다릅니다.

주: Person_t 유형에서 작성된 유형화 테이블에는 구조화 유형 Address_t의 Address 컬럼이 있습니다. 모든 구조화 유형 컬럼은 그 컬럼의 구조화 유형에 대해 변환 함수를 정의해야 합니다. 변환 함수 정의에 대한 내용은 359 페이지의 『호스트 언어 프로그램에 대한 맵핑 작성: 변환 함수』를 참조하십시오.

결국 Manager_t와 Architect_t는 둘다 Employee_t의 부속 유형입니다. 이는 Employee_t의 모든 속성을 계승하고 그 유형에 맞도록 확장합니다. 그러므로, 유형 Manager_t의 인스턴스에는 Name, Age, Address, SerialNum, Salary, Dept Bonus 등의 7가지 속성이 있습니다.

참조 유형 및 그 표현 유형

작성한 모든 구조화 유형에 대해 DB2는 자동으로 동반 유형을 작성합니다. 동반 유형은 *참조 유형*이라고 하며, 참조하는 구조화 유형은 *참조된 유형*이라고 합니다. 유형화 테이블은 329 페이지의 『유형화 테이블에서의 구조화 유형 사용』에 설명

된 대로 참조 유형을 특별한 용도로 사용할 수 있게 합니다. SQL문에서 다른 사용자 정의 유형과 같이 참조 유형을 사용할 수도 있습니다. SQl문에서 참조 유형을 사용하려면 REF(*type-name*)을 사용하십시오. 여기서 *type-name*은 참조된 유형을 나타냅니다.

DB2는 참조 유형을 유형화 테이블에 있는 오브젝트 식별자 컬럼의 유형으로 사용합니다. 오브젝트 식별자는 유형화 테이블 계층에 있는 행 오브젝트를 고유하게 식별합니다. DB2는 또한 참조 유형을 사용하여 참조를 유형화 테이블에 있는 행에 저장합니다. 참조 유형을 사용하여 테이블에 있는 각 오브젝트를 참조할 수 있습니다. 참조 사용에 대한 자세한 내용은 334 페이지의 『참조 유형 사용』을 참조하십시오. 유형화 테이블에 대한 자세한 내용은 324 페이지의 『유형화 테이블에서의 오브젝트 저장』을 참조하십시오.

참조는 명백하게 유형 지정됩니다. 그러므로 표현식에서 유형을 사용할 수 있어야 합니다. 유형 계층의 루트 유형을 작성할 때 CREATE TYPE문의 REF USING 절을 사용하여 참조의 기본 유형을 지정할 수 있습니다. 참조 기본 유형은 *표현 유형*이라고 합니다. REF USING절을 사용하여 표현 유형을 지정하지 않으면, DB2는 VARCHAR(16) FOR BIT DATA의 기본 데이터 유형을 사용합니다. 루트 유형의 표현 유형은 그 모든 부속 유형에서 계승됩니다. REF USING절은 계층의 루트 유형을 정의할 때만 유효합니다. 이 절 전반에 걸쳐 사용된 예에서 `BusinessUnit_t` 유형의 표현 유형은 INTEGER인 반면 `Person_t`의 표현 유형을 VARCHAR(13)입니다.

참조 유형 유형변환 및 비교

DB2는 양쪽 디렉토리에서 참조 유형과 그 표현 유형 간에 값을 유형변환하는 함수를 자동으로 작성합니다. CREATE TYPE문에는 *SQL 참조서*에서 설명된 대로 이런 두 유형변환 함수의 이름을 선택할 수 있게 하는 선택적 CAST WITH절이 있습니다. 기본적으로, 유형변환 함수의 이름은 구조화 유형과 그 참조 표현 유형의 이름과 같습니다. 예를 들어, 317 페이지의 『구조화 유형 계층 작성』의 CREATE TYPE Person_t문은 다음 함수를 자동으로 작성합니다.

```
CREATE FUNCTION VARCHAR(REF(Person_t))
   RETURNS VARCHAR
```

DB2는 조작을 반전시키는 함수를 작성하기도 합니다.

```
CREATE FUNCTION Person_t(VARCHAR(13))
   RETURNS REF(Person_t)
```

새 값을 유형화 테이블에 삽입하거나 참조 값을 다른 값과 비교할 때마다 이 유형변환 함수를 사용합니다.

DB2는 또한 =, <>, <, <=, > 그리고 >=과 같은 비교 연산자를 사용하여 참조 유형을 비교하는 함수를 작성합니다. 참조 유형의 비교 연산자에 대한 자세한 정보는 *SQL 참조서*를 참조하십시오.

기타 시스템-생성 루틴

작성하는 모든 구조화 유형은 DB2에서 구조화 유형 값을 구성, 관찰 또는 수정할 때 사용하는 함수 및 메소드 세트를 암시적으로 작성하게 합니다. 즉, 예를 들어 Person_t 유형에 대해 DB2는 유형을 작성할 때 다음 함수와 메소드를 자동으로 작성합니다.

구성자 함수

유형과 같은 이름의 함수가 작성됩니다. 이 함수에는 매개변수가 없으며 그 모든 속성이 널(NULL)로 설정된 유형의 인스턴스를 리턴합니다. 예를 들어, Person_t에 대해 작성된 함수는 다음 명령문이 실행된 것과 같습니다.

```
CREATE FUNCTION Person_t ( ) RETURNS Person_t
```

Manager_t 부속 유형에 대해 구성자 함수는 다음 명령문이 실행된 것처럼 작성됩니다.

```
CREATE FUNCTION Manager_t ( ) RETURNS Manager_t
```

컬럼에 삽입할 유형의 인스턴스를 구성하려면 변환 메소드와 함께 구성자 함수를 사용하십시오. 유형이 컬럼 대신 테이블에 저장되면 구성자 함수를 변환 메소드에서 사용하여 유형의 인스턴스를 삽입하지 않아도 됩니다. 데이터를 유형화 테이블에 삽입하는 것에 대한 자세한 내용은 351 페이지의 『구조화 유형 값이 있는 행 삽입』을 참조하십시오.

변환 메소드

변환 메소드는 오브젝트의 속성마다 존재합니다. 메소드가 호출되는 유형의 인스턴스는 메소드의 *주제* 인스턴스라고 합니다. 주제 인스턴스에서 호

출된 변환 메소드가 속성에 대해 새 값을 수신하면 메소드는 새 값으로 갱신된 속성과 함께 새 인스턴스를 리턴합니다. 그러므로, Person_t 유형에 대해 DB2는 name, age 그리고 address 속성마다 변환 메소드를 작성합니다.

예를 들어, DB2가 age 속성에 대해 작성하는 변환 메소드는 마치 다음 명령문이 실행된 것과 같습니다.

```
ALTER TYPE Person_t
   ADD METHOD AGE(int)
   RETURNS Person_t;
```

오브젝트 변환에 대한 자세한 내용은 353 페이지의 『구조화 유형 값의 검색 및 수정』을 참조하십시오.

관찰 메소드

관찰 메소드는 오브젝트의 각 속성마다 존재합니다. 속성에 대한 메소드가 예상 유형이나 부속 유형의 오브젝트를 수신하면 메소드를 그 오브젝트에 대해 속성의 값을 리턴합니다.

예를 들어, DB2가 Person_t 유형의 age 속성에 대해 작성하는 관찰 메소드는 DB2에서 다음 명령을 발행한 것과 같습니다.

```
ALTER TYPE Person_t
   ADD METHOD AGE()
   RETURNS INTEGER;
```

관찰 메소드 사용에 대한 자세한 내용은 353 페이지의 『구조화 유형 값의 검색 및 수정』을 참조하십시오.

구조화 유형에서 메소드를 호출하려면 메소드 호출 연산자 ‘..’를 사용하십시오. 메소드 호출에 대한 자세한 내용은 *SQL 참조서*를 참조하십시오.

유형 동작 정의

구조화 유형의 동작을 정의하려면 사용자 정의 메소드를 작성할 수 있습니다. 구별 유형에 대해 메소드를 작성할 수 없습니다. 메소드 작성은 루틴을 작성하는 것과 비슷합니다. 단, 그 메소드를 특히 한 유형에 대해 작성되므로 그 유형과 동작은 밀접하게 통합되어 있습니다.

메소드 스펙은 CREATE METHOD문을 발행하기 전에 유형에 연관되어 있어야 합니다. 다음 명령문은 calc_bonus라고 하는 메소드에 대한 메소드 스펙을 Employee_t 유형에 추가합니다.

```
ALTER TYPE Employee_t
   ADD METHOD calc_bonus (rate DOUBLE)
   RETURNS DECIMAL(7,2)
   LANGUAGE SQL
   CONTAINS SQL
   NO EXTERNAL ACTION
   DETERMINISTIC;
```

일단 메소드 스펙을 유형에 연관시키면 메소드를 매소드 스펙에 따라 외부 메소드 또는 SQL 본문 메소드로 작성하여 유형의 동작을 정의할 수 있습니다. 예를 들어, 다음 명령문은 Employee_t 유형과 같은 스키마에 상주하는 calc_bonus라는 SQL 메소드를 등록합니다.

```
CREATE METHOD calc_bonus (rate DOUBLE)
   FOR Employee_t
   RETURN SELF..salary * rate;
```

calc_bonus라는 메소드를 원하는 대로 작성할 수 있는데, 매개변수의 수 또는 유형이 다르거나 다른 유형 계층에 있는 유형에 대해 정의되어야 합니다. 다시 말해서 Architect_t에 대해 매개변수 유형과 같은 수의 매개변수를 가진 calc_bonus라는 또다른 메소드를 작성할 수 없습니다.

주: DB2는 현재 동적 디스패치를 지원하지 않습니다. 즉, 한 유형에 대해 메소드를 선언한 다음 동일한 수의 매개변수를 사용하여 부속 유형에 대해 메소드를 다시 정의할 수 없습니다. 일시적인 해결책으로 TYPE 술어를 사용하여 동적 유형을 결정한 다음, TREAT AS절을 사용하여 각 동적 유형에 대해 다른 메소드를 호출할 수 있습니다. 부속 유형을 처리하는 변환 함수의 예는 372 페이지의 『DB2에서 부속 유형 데이터 검색(바인드아웃)』에서 참조하십시오.

메소드 등록, 작성 그리고 호출에 대해서는 409 페이지의 『제14장 사용자 정의 함수(UDFs)와 메소드』 및 433 페이지의 『제15장 사용자 정의 함수(UDF)』에서 참조하십시오.

유형화 테이블에서의 오브젝트 저장

구조화 유형의 인스턴스를 유형의 각 속성이 별도의 컬럼에 저장되는 유형화 테이블에서 행으로 저장하거나, 유형의 모든 속성이 단일 컬럼에 저장되는 컬럼에서 오브젝트로 저장할 수 있습니다. 유형화 테이블에는 식별의 속성이 있습니다. 즉, 다른 테이블에서 참조를 사용하여 인스턴스의 속성을 액세스할 수 있습니다. 다른 테이블에서 인스턴스를 참조하려면 유형화 테이블을 사용해야 합니다. 오브젝트를 다른 테이블에서 식별하지 않아도 되면 오브젝트를 컬럼에 저장하십시오.

오브젝트가 테이블 행으로 저장되면 테이블의 각 컬럼에는 오브젝트의 속성이 하나씩 들어 있습니다. 예를 들어, '사람' 인스턴스를 '이름' 컬럼과 '나이' 컬럼이 있는 테이블에 저장할 수 있습니다. 다음은 Person의 인스턴스를 저장하는 CREATE TABLE문의 예입니다.

```
CREATE TABLE Person OF Person_t
   (REF IS Oid USER GENERATED)
```

Person의 인스턴스를 테이블에 삽입하려면 다음 구문을 사용하십시오.

```
INSERT INTO Person (Oid, Name, Age)
   VALUES(Person_t('a'), 'Andrew', 29);
```

표 10. Person 유형화 테이블

Oid	이름	나이	주소
a	Andrew	29	

프로그램은 유형화 테이블의 컬럼을 액세스하여 오브젝트의 속성을 액세스합니다.

UPDATE Person SET Age=30 WHERE Name='Andrew';

이전 UPDATE문 이후 테이블은 다음과 같습니다.

표 11. 갱신 후의 Person 유형화 테이블

Oid	이름	나이	주소
a	Andrew	30	

Employee_t라는 Person_t 서브테이블이 있으므로, Employee_t 인스턴스는 Person 테이블에 저장될 수 없고 다른 테이블에 저장되어야 합니다. 이 테이블을 *서브테이블*이라고 합니다. 다음 REATE TABLE문은 Person 테이블 아래에 Employee 서브테이블을 작성합니다.

```
CREATE TABLE Employee OF Employee_t UNDER Person
   INHERIT SELECT PRIVILEGES
   (SerialNum WITH OPTIONS NOT NULL,
   Dept WITH OPTIONS SCOPE BusinessUnit);
```

그리고 Employee 테이블에 다시 삽입하면 다음과 같습니다.

```
INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary)
   VALUES (Employee_t('s'), 'Susan', 39, 24001, 37000.48)
```

표 12. Employer 유형화 서브테이블

Oid	이름	나이	주소	SerialNum	Salary	Dept
s	Susan	39		24001	37000.48	

다음 조회를 실행하면 Susan의 정보가 리턴됩니다.

```
SELECT *
   FROM Employee
   WHERE Name='Susan';
```

이런 두 테이블에 대해 흥미로운 것은 Person 테이블에서 SQL문을 실행하는 것만으로도 '직원'과 '사람' 모두에 대한 인스턴스를 액세스할 수 있다는 것입니다. 이 기능을 *대체 가능성*이라고 하고 328 페이지의 『구조화 유형의 추가 특성』에서 설명됩니다. 유형 계층에서 더 높은 위치에 있는 인스턴스를 가지고 있는 테이블을 조회하면, 계층에서 더 낮은 위치에 있는 유형의 인스턴스를 자동으로 얻게 됩니다. 다시 말해 Person 테이블은 논리적으로 SELECT, UPDATE 및 DELETE 문인 것처럼 표시됩니다.

표 13. Person 테이블에는 Person과 Employee 인스턴스가 있습니다.

Oid	이름	나이	주소
a	Andrew	30	(널(NULL))
s	Susan	39	(널(NULL))

다음 조회를 실행하면 (사람) Andrew와 (직원) Susan에 대한 오브젝트 식별자 및 Person_t 정보를 얻게 됩니다.

```
SELECT *
FROM Person;
```

대체 가능성에 대한 자세한 내용은 328 페이지의 『구조화 유형의 추가 특성』을 참조하십시오.

유형화 테이블에서의 오브젝트간 관계 정의

유형화 테이블 하나의 오브젝트와 다른 테이블의 오브젝트간 관계를 정의할 수 있습니다. 동일한 유형화 테이블의 오브젝트간 관계를 정의할 수도 있습니다. 예를 들어, 부서의 인스턴스가 들어 있는 유형화 테이블을 정의했다고 가정하십시오. Employee 테이블에서 부서 번호를 유지보수하는 대신, Employee 테이블의 Dept 컬럼에는 BusinessUnit 테이블에 있는 부서 중 하나에 대한 논리적 포인터가 들어 있습니다. 이런 포인터를 *참조*라고 하며 그림9에서 설명됩니다.

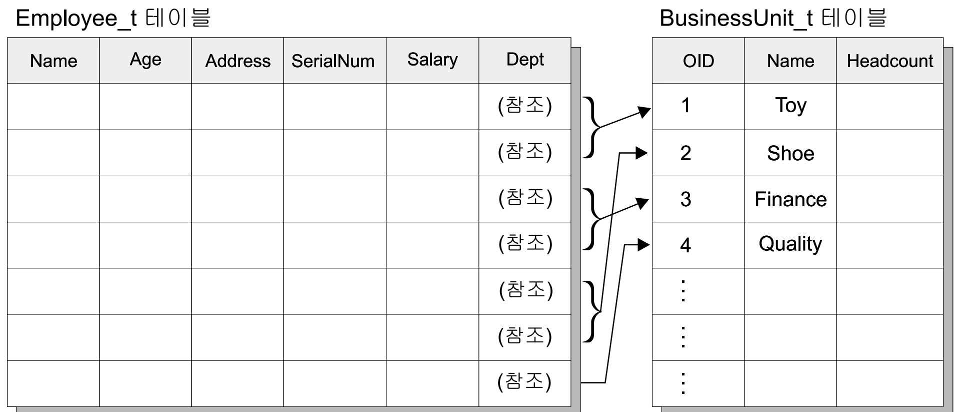


그림 9. Employee_t에서 BusinessUnit_t로의 구조화 유형 참조

중요사항: 참조는 참조 제약조건과 동일한 기능을 수행하지 않습니다. 존재하지 않는 부서를 참조할 수도 있습니다. 부서와 직원간에 무결성을 유지보수하는 것이 중요하면, 이런 두 테이블간에 참조 제한조건을 정의할 수 있습니다. 참조의 실제 능력은 테이블간의 관계를 탐색하는 조회를 작성할 수 있는 기능을 제공한다는 데 있습니다. 조회는 관계를 참조 해제하고 포인트된 오브젝트를 인스턴스화합니다. 이 작업을 수행할 때 사용한 연산자는 *참조 해제* 연산자라고 하며, ->와 같습니다.

예를 들어, Employee 테이블에서 다음 조회는 참조 해제 연산자를 사용하여 DB2에게 Dept 컬럼에서 BusinessUnit 테이블로의 경로를 따르도록 지시합니다. 참조 해제 연산자는 Name 컬럼의 값을 리턴합니다.

```
SELECT Name, Salary, Dept->Name
   FROM Employee;
```

유형화 테이블에서 조회를 작성하는 것에 대한 자세한 정보는 342 페이지의『유형화 테이블 조회』를 참조하십시오.

컬럼에서의 오브젝트 저장

컬럼에 오브젝트를 저장하면 DB2 내장 데이터 유형으로 적절히 모델화될 수 없는 업무 오브젝트에 대한 사실을 모델화할 때 유용합니다. 다시 말해서, 업무 오브젝트(직원, 부서 등)를 유형화 테이블에 저장할 수 있지만, 이 오브젝트에는 구조화 유형을 사용하여 가장 잘 모델화되는 속성이 있을 수 있습니다.

예를 들어, 응용프로그램이 주소의 특정 부분을 액세스해야 한다고 가정하십시오. 주소를 구조화되지 않은 문자열로 저장하는 대신, 이를 그림10에서와 같이 구조화 오브젝트로 저장할 수 있습니다.

Person

Name (VARCHAR)	Age (INT)	Address (Address_t)			
		Street	Number	City	State

그림 10. 구조화 유형으로서의 주소 속성

또한, 주소의 유형 계층을 정의하여 여러 나라에서 사용되는 서로 다른 주소 형식을 모델화할 수 있습니다. 예를 들어, 우편번호 코드가 있는 미국 주소 유형과 이웃 속성이 필요한 브라질 주소 유형을 둘다 포함시킬 수 있습니다. Address_t 유형 계층은 349 페이지의『컬럼에 구조화 유형 인스턴스 삽입』에 정의되어 있습니다.

오브젝트가 컬럼 값으로 저장되면, 속성은 테이블의 행에 저장된 오브젝트와 같이 외부적으로 표현되지 않습니다. 대신 메소드를 사용하여 그 속성을 처리해야 합니다. DB2는 *관찰* 메소드를 생성하여 속성을 리턴하고, *변환* 메소드를 생성하여 속성을 변경합니다. 다음

예는 관찰 메소드 하나와 변환 메소드 두 개를 사용합니다. 하나는 Number 속성을 위한 것이고, 다른 하나는 Street 속성을 위한 것으로 주소를 변경합니다.

```
UPDATE Employee
   SET Address=Address..Number('4869')..Street('Appletree')
   WHERE Name='Franky'
   AND Address..State='CA';
```

앞의 예에서 UPDATE문의 SET절은 Number와 Street 변환 메소드를 호출하여 Address_t 유형의 인스턴스 속성을 갱신합니다. WHERE절은 술어가 두 개인 갱신 명령문 조작을 제한합니다. 하나는 Name 컬럼의 동등성 비교이고, 다른 하나는 Address 컬럼의 State 관찰 메소드를 호출하는 동등성 비교입니다.

구조화 유형의 추가 특성

대체 가능성

SELECT, UPDATE 또는 DELETE문이 유형화 테이블에 적용되면 조작은 명명된 테이블과 그 모든 서브테이블에 적용됩니다. 예를 들어, Person_t에서 유형화 테이블을 작성하고 그 테이블의 모든 행을 선택하면, 응용프로그램은 Person 유형의 인스턴스뿐만 아니라 Employee 부속 유형과 기타 부속 유형의 인스턴스에 대한 Person 정보를 받습니다. 대체 가능성의 특성은 또한 부속 유형에서 작성된 서브테이블에 적용됩니다. 예를 들어, Employee 서브테이블의 SELECT, UPDATE 및 DELETE문은 Employee_t 유형과 그 부속 유형에 적용됩니다.

이와 비슷하게 Address_t 유형으로 정의된 컬럼에는 미국 주소 또는 브라질 주소의 인스턴스가 있습니다.

이와 달리 INSERT 조작은 INSER문에서 지정된 테이블에만 적용됩니다. Employee 테이블에 삽입하면, Employee_t 오브젝트가 Person 테이블 계층에 작성됩니다.

구조화 유형을 매개변수로 함수에 전달하거나 함수에서 결과로 전달하면 부속 유형 인스턴스를 대체할 수도 있습니다. 스칼라 함수에 Address_t 유형의 매개변수가 있으면, US_addr_t와 같은 그 부속 유형 중 하나의 인스턴스를 Address_t의 인스턴스 대신 전달할 수 있습니다. 테이블 함수는 구조화 유형 컬럼을 리턴할 수 없습니다.

컬럼이나 테이블은 하나의 유형으로 정의되지만 다른 유형의 인스턴스가 들어 있을 수도 있으므로, 정의에 사용된 유형과 실제 런타임시 리턴되는 인스턴스 유형을 구분하는 것이 중요합니다. 컬럼, 행 또는 함수 매개변수의 구조화 유형의 정의를 *정적 유형*이라고 합니다. 구조화 유형 인스턴스의 실제 유형을 *동적 유형*이라고 합니다. 동적 유형에 대한 정보를 검색하려면, 응용프로그램은 344 페이지의 『기타 유형-관련 내장 함수』에서 설명된 TYPE_NAME, TYPE_SCHEMA 및 TYPE_ID의 내장 함수를 사용할 수 있습니다.

인스턴스화 가능성

유형은 *INSTANTIABLE* 또는 *NOT INSTANTIABLE*로 정의될 수도 있습니다. 기본적으로 유형은 인스턴스화될 수 있습니다. 즉 그 오브젝트의 인스턴스를 작성할 수 있습니다. 다른 한편, 인스턴스화되지 않는 유형은 유형 계층에서 추가 조절을 위한 모델의 역할을 합니다. 예를 들어, NOT INSTANTIABLE절을 사용하여 Person_t를 정의하면, 데이터베이스에 ’사람’ 인스턴스를 저장할 수 없으며 Person_t를 사용하여 테이블을 작성하거나 볼 수 없습니다. 대신 정의한 Employee_t의 인스턴스나 Person_t의 기타 부속 유형만 저장할 수 있습니다.

유형화 테이블에서의 구조화 유형 사용

유형화 테이블 작성

유형화 테이블은 특성이 CREATE TYPE문으로 정의된 오브젝트의 인스턴스를 실제로 저장하기 위해 사용됩니다. CREATE TABLE문의 변형을 사용하여 유형화 테이블을 작성할 수 있습니다. 구조화된 유형의 계층 구조를 기반으로 입력된 테이블의 계층 구조를 작성할 수도 있습니다. 데이터베이스 테이블에 부속 유형의 인스턴스를 저장하려면 해당 테이블 계층을 작성해야 합니다.

다음 예는 326 페이지의 그림9에 설명된 유형 계층을 기반으로 테이블 계층을 작성하는 방법을 보여줍니다.

여기에는 BusinessUnit 유형화 테이블을 작성하는 SQL이 있습니다.

```
CREATE TABLE BusinessUnit OF BusinessUnit_t
(REF IS Oid USER GENERATED);
```

여기에는 Person 테이블 계층에서 테이블을 작성하는 SQL이 있습니다.

```
CREATE TABLE Person OF Person_t
(REF IS Oid USER GENERATED);

CREATE TABLE Employee OF Employee_t UNDER Person
   INHERIT SELECT PRIVILEGES
   (SerialNum WITH OPTIONS NOT NULL,
   Dept WITH OPTIONS SCOPE BusinessUnit );

CREATE TABLE Student OF Student_t UNDER Person
   INHERIT SELECT PRIVILEGES;

CREATE TABLE Manager OF Manager_t UNDER Employee
   INHERIT SELECT PRIVILEGES;

CREATE TABLE Architect OF Architect_t UNDER Employee
   INHERIT SELECT PRIVILEGES;
```

테이블의 유형 정의

이전 예에서 작성된 첫번째 유형화 테이블은 BusinessUnit입니다. 이 테이블은 BusinessUnit_t 유형으로 정의되었으므로, 해당 유형의 인스턴스를 보유합니다. 즉, BusinessUnit_t 구조화 유형의 각 속성에 해당하는 컬럼이 있으며, *오브젝트 식별자 컬럼*이라는 추가 컬럼이 하나 있습니다.

오브젝트 식별자 명명

유형화 테이블에는 다른 오브젝트로 참조될 수 있는 오브젝트가 들어 있으므로, 모든 유형나 테이블은 첫번째 컬럼으로 *오브젝트 식별자* 컬럼을 사용합니다. 이 예에서 오브젝트 식별자 컬럼의 유형은 REF(BusinessUnit_t)입니다. REF IS ... USER GENERATED절을 사용하여 오브젝트 식별자 컬럼을 명명할 수 있습니다. 이 경우 컬럼의 이름은 Oid입니다. REF IS절의 USER GENERATED 부분은 새롭게 삽입된 모든 행의 오브젝트 식별자에 대해 초기 값을 제공해야 함을 나타냅니다. 오브젝트 식별자를 삽입한 후 오브젝트 식별자의 값을 갱신할 수 없습니다. 오브젝트 식별자를 자동으로 생성하도록 DB2를 구성에 대해서는 347 페이지의 『시스템 생성 오브젝트 식별자 정의』를 참조하십시오.

테이블 계층에서 위치 지정

Person 유형화 테이블은 Person_t 유형입니다. 직원과 학생의 부속 유형 인스턴스를 저장하려면, Person 테이블, Employee 및 Student의 서브테이블을 작성해야 합니다. 두 개의 Employee_t 추가 부속 유형도 테이블이 필요합니다. 이 서브

테이블의 이름은 Manager와 Architect입니다. 부속 유형이 상위 유형의 속성을 계승하는 것처럼 서브테이블은 오브젝트 식별자 컬럼을 포함한 그 수퍼 테이블의 컬럼을 계승합니다.

주: 서브테이블은 수퍼 테이블과 같은 스키마에 상주해야 합니다.

그러므로 Employee 서브테이블의 행은 Oid, Name, Age, Address, SerialNum, Salary 및 Dept 등 총 7개의 컬럼이 있습니다.

수퍼 테이블에서 조작되는 SELECT, UPDATE 또는 DELETE문은 모든 하위 테이블에서도 자동으로 조작됩니다. 예를 들어, Employee 테이블에서 UPDATE문은 Employee, Manager 및 Architect 테이블에 있는 행에 영향을 줄 수 있지만, Manager 테이블의 UPDATE문은 Manager 행에만 영향을 줄 수 있습니다.

SELECT, INSERT 또는 DELETE문이 테이블만 지정하도록 제한하려면 345 페이지의 『ONLY를 사용하여 특정 유형의 오브젝트 리턴』에 설명된 대로 ONLY 옵션을 사용하십시오.

SELECT 특권 계승 지정

CREATE TABLE문의 INHERIT SELECT PRIVILEGES절은 Employee와 같이 결과의 서브테이블을 지정하고, *초기*에 UNDER절을 사용하여 작성된 Person과 같은 수퍼 테이블과 같은 사용자와 그룹에서 액세스될 수 있습니다. 수퍼 테이블에서 현재 SELECT 특권을 보유 중인 사용자나 그룹에게는 새롭게 작성된 서브테이블의 SELECT 특권이 부여됩니다. 서브테이블 작성자는 SELECT 특권의 권한 준 사용자입니다. 서브테이블에서 DELETE와 UPDATE와 같은 특권을 지정하려면, 일반 테이블에서 특권을 지정할 때 사용한 것과 같은 명시적 GRANT 또는 REVOKE문을 발행해야 합니다. INHERIT SELECT PRIVILEGES절에 대한 자세한 내용은 *SQL 참조서*에서 참조하십시오.

특권은 테이블 계층의 모든 레벨에서 별도로 권한 부여되고 권한 취소되어야 합니다. 서브테이블을 작성하면 그 서브테이블에서 계승된 SELECT 특권을 다시 호출할 수도 있습니다. 서브테이블에서 계승된 SELECT 특권을 재호출하면, 수퍼 테이블에 대해 SELECT 특권이 있는 사용자가 서브테이블에만 나타나는 컬럼을 볼 수 없게 막을 수 있습니다. 서브테이블에서 계승된 SELECT 특권을 재호출하면, 수퍼 테이블에 대해 SELECT 특권만 있는 사용자가 서브테이블 행의 수퍼 테이

블 컬럼을 볼 수 없도록 제한할 수 있습니다. 사용자는 수퍼 테이블에 필요한 특권을 가지고 있으면 그 수퍼 테이블에서만 직접 작업할 수 있습니다. 그러므로, 사용자가 서브테이블에서 관리자의 보너스를 선택하지 못하도록 막으려면, 그 테이블에 대해 SELECT 특권을 재호출하고 이 정보가 필요한 사용자에게만 이를 부여하십시오.

컬럼 옵션 정의

WITH OPTIONS절을 사용하면 유형화 테이블에서 개별 컬럼에 적용시킬 수 있는 옵션을 정의할 수 있습니다. WITH OPTIONS의 형식은 다음과 같습니다.

```
column-name WITH OPTIONS column-options
```

여기서 *column-name*은 CREATE TABLE 또는 ALTER TABLE문에 있는 컬럼의 이름을 나타내고, *column-options*은 컬럼에 대해 정의된 옵션을 나타냅니다.

예를 들어, 사용자가 SerialNum 컬럼에 널(NULL)을 삽입하지 못하도록 막으려면 다음과 같이 NOT NULL 컬럼 옵션을 지정하십시오.

```
(SerialNum WITH OPTIONS NOT NULL)
```

참조된 컬럼의 범위 정의

WITH OPTIONS는 컬럼의 SCOPE를 지정할 때 사용될 수도 있습니다. 예를 들어, Employee 테이블과 그 서브테이블에서 절은 다음과 같습니다.

```
Dept WITH OPTIONS SCOPE BusinessUnit
```

이 테이블의 Dept 컬럼과 그 서브테이블에 BusinessUnit의 *범위*가 있음을 선언합니다. 즉, Employee 테이블에서 이 컬럼의 참조 값은 BusinessUnit 테이블의 오브젝트를 참조하기 위한 것입니다.

예를 들어, Employee 테이블에서 다음 조회는 참조 해제 연산자를 사용하여 DB2에게 Dept 컬럼에서 BusinessUnit 테이블로의 경로를 따르도록 지시합니다. 참조 해제 연산자는 Name 컬럼의 값을 리턴합니다.

```
SELECT Name, Salary, Dept->Name
   FROM Employee;
```

참조 및 참조 범위에 대한 자세한 내용은 334 페이지의『참조 유형 사용』을 참조하십시오.

유형화 테이블에서 데이터 처리 중

이전 예에서 구조화 유형을 작성하고 해당 테이블과 서브테이블을 작성한 다음, 데이터베이스의 구조는 그림11과 같습니다.

그림 11. 유형화 테이블 계층

계층이 설정되면 INSERT문을 사용하여 테이블에서 데이터를 처리할 수 있습니다. 단, 오브젝트 식별자 컬럼에서 데이터를 처리해야 하며, 선택적으로 각 테이블이나 서브테이블에 있는 오브젝트의 추가 속성 데이터를 처리해야 합니다. 오브젝트 식별자 컬럼은 명백한 유형 지정인 REF 유형이므로, 구조화 유형을 작성했을 때 시스템에서 생성한 유형변환 함수를 사용하여 사용자가 제공한 오브젝트 식별자 값을 유형변환해야 합니다.

```
INSERT INTO BusinessUnit (Oid, Name, Headcount)
   VALUES(BusinessUnit_t(1), 'Toy', 15);

INSERT INTO BusinessUnit (Oid, Name, Headcount)
   VALUES(BusinessUnit_t(2), 'Shoe', 10);

INSERT INTO Person (Oid, Name, Age)
   VALUES(Person_t('a'), 'Andrew', 20);

INSERT INTO Person (Oid, Name, Age)
   VALUES(Person_t('b'), 'Bob', 30);

INSERT INTO Person (Oid, Name, Age)
   VALUES(Person_t('c'), 'Cathy', 25);

INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary, Dept)
   VALUES(Employee_t('d'), 'Dennis', 26, 105, 30000, BusinessUnit_t(1));

INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary, Dept)
   VALUES(Employee_t('e'), 'Eva', 31, 83, 45000, BusinessUnit_t(2));

INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary, Dept)
   VALUES(Employee_t('f'), 'Franky', 28, 214, 39000, BusinessUnit_t(2));
```

```
INSERT INTO Student (Oid, Name, Age, SerialNum, GPA)
   VALUES(Student_t('g'), 'Gordon', 19, '10245', 4.7);

INSERT INTO Student (Oid, Name, Age, SerialNum, GPA)
   VALUES(Student_t('h'), 'Helen', 20, '10357', 3.5);

INSERT INTO Manager (Oid, Name, Age, SerialNum, Salary, Dept, Bonus)
   VALUES(Manager_t('i'), 'Iris', 35, 251, 55000, BusinessUnit_t(1), 12000);

INSERT INTO Manager (Oid, Name, Age, SerialNum, Salary, Dept,
Bonus)
   VALUES(Manager_t('j'), 'Christina', 10, 317, 85000, BusinessUnit_t(1),
25000);

INSERT INTO Manager (Oid, Name, Age, SerialNum, Salary, Dept, Bonus)
   VALUES(Manager_t('k'), 'Ken', 55, 482, 105000, BusinessUnit_t(2), 48000);

INSERT INTO Architect (Oid, Name, Age, SerialNum, Salary, Dept, StockOption)
   VALUES(Architect_t('l'), 'Leo', 35, 661, 92000, BusinessUnit_t(2), 20000);
```

이전 예에서는 주소를 삽입하지 않습니다. 구조화 유형 값을 컬럼에 삽입하는 방법에 대한 내용은 351 페이지의『구조화 유형 값이 있는 행 삽입』을 참조하십시오.

행을 유형화 테이블에 삽입할 때 삽입된 행에서 첫번째 값은 테이블에 삽입되는 데이터의 오브젝트 식별자여야 합니다. 또한, 비유형화 테이블에서와 같이 NOT NULL로 정의된 모든 컬럼에 데이터를 제공해야 합니다. 마지막으로, 적절한 유형의 모든 참조 값 표현식을 사용하여 참조 속성을 초기화할 수 있음에 유의하십시오. 이전 예에서 직원의 Dept 참조는 적절한 유형변환 상수로 입력됩니다. 그러나 부속 조회를 사용하여 다음 예에서와 같이 참조를 획득할 수도 있습니다.

```
INSERT INTO Architect (Oid, Name, Age, SerialNum, Salary, Dept, StockOption)
   VALUES(Architect_t('m'), 'Brian', 7, 882, 112000,
   (SELECT Oid FROM BusinessUnit WHERE name = 'Toy'), 30000);
```

참조 유형 사용

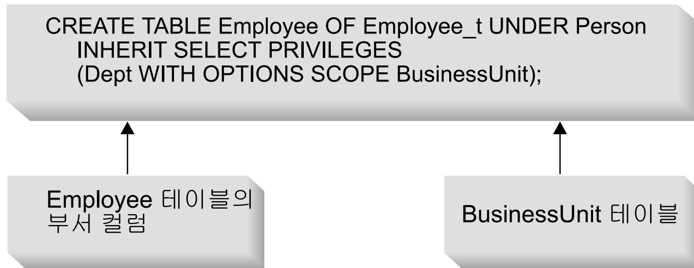
각각의 구조화 유형에 대해 DB2는 해당 참조 유형을 지원합니다. 예를 들어, Person_t 유형을 작성하면 DB2는 자동으로 REF(Person_t)의 유형을 작성합니다. REF(Person_t) 유형(및 Person_t의 모든 부속 유형의 REF 유형)은 기본적으로 VARCHAR (16) FOR BIT DATA이지만 CREATE TYPE문에 대해 REF USING절을 사용하여 다른 표현 유형을 선택할 수 있습니다. 그 참조 유형은 구조화 유형의 인스턴스를 저장하기 위해 작성하는 유형화 테이블의 오브젝트 식별자 컬럼의 기본입니다. 예를 들어, 참조 유형의 기본 표현 유형을 사용하여 People_t 루트 유형을 작성하면 연관된 People 테이블의 오브젝트 식별자 컬럼은 VARCHAR(16) FOR BIT DATA를 기반으로 합니다.

참조 유형 비교

참조 유형은 명백하게 유형 지정됩니다. 참조를 상수에 비교하려면, 상수를 적절한 참조 유형으로 유형변환하거나 참조 유형을 기본 유형으로 유형변환한 후 비교를 수행할 수 있습니다. 주어진 유형 계층에서 모든 참조는 동일한 참조 표현 유형이 있습니다. S와 T에 공통 상위 유형이 있다는 조건을 통해 REF(S)와 REF(T)를 비교할 수 있습니다. 오브젝트 식별자 컬럼의 고유함은 테이블에서만 강요되므로, 하나의 테이블 계층에 있는 REF(T)의 값과 다른 테이블 계층에 있는 REF(T)의 값은 서로 다른 행을 참조해도 같을 수 있습니다.

참조를 사용한 의미 관계 정의

CREATE TABLE의 WITH OPTIONS절을 사용하여 테이블 하나에 있는 컬럼과 같은 테이블 또는 다른 테이블에 있는 오브젝트간에 존재하는 관계를 정의할 수 있습니다. 예를 들어, 336 페이지의 그림12에서와 같이 BusinessUnit와 Person 테이블 계층에서 각 직원의 부서는 실제로 BusinessUnit에 있는 오브젝트를 참조합니다. 주어진 참조 컬럼의 목적지 오브젝트를 정의하려면 WITH OPTIONS 절에서 SCOPE 키워드를 사용하십시오.



Employee (및 서브테이블)

Oid	Name	Age	Address	SerialNum	Salary	Dept

BusinessUnit

Oid	Name	Age	Headcount

그림 12. BusinessUnit 오브젝트를 참조하는 Dept 속성

자체-참조 관계: 같은 유형화 테이블에서도 오브젝트에 대해 범위가 정해진 참조를 정의할 수 있습니다. 다음 예에 있는 명령문은 부분에 대해 유형화 테이블을 하나 작성하고, 공급자에 대해 유형화 테이블을 하나 작성합니다. 참조 유형 정의를 표시하기 위해 샘플은 유형을 작성할 때 사용되는 명령문을 포함하기도 합니다.

```
CREATE TYPE Company_t AS
   (name VARCHAR(30),
   location VARCHAR(30))
   MODE DB2SQL;

CREATE TYPE Part_t AS
   (Descript VARCHAR(20),
   Supplied_by REF(Company_t),
   Used_in REF(part_t))
   MODE DB2SQL;

CREATE TABLE Suppliers OF Company_t
```

```
   (REF IS suppno USER GENERATED);

CREATE TABLE Parts OF Part_t
   (REF IS Partno USER GENERATED,
   Supplied_by WITH OPTIONS SCOPE Suppliers,
   Used_in WITH OPTIONS SCOPE Parts);
```

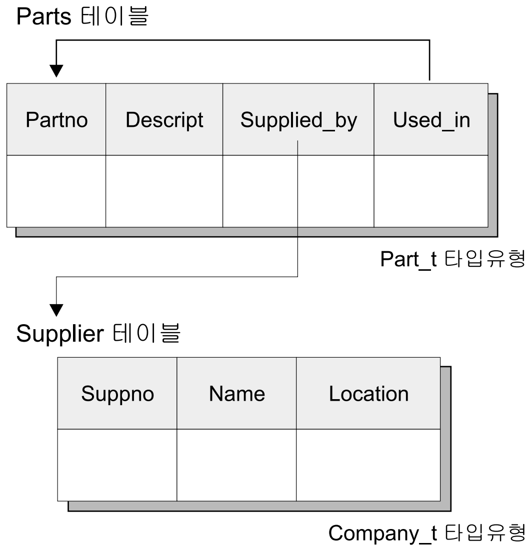


그림 13. 자체 참조 범위의 예

범위 지정된 참조 없이 외부 조인이나 상관 부속 조회로 작성되었을 때, 조회는 범위 지정된 참조를 사용하여 작성할 수 있습니다. 자세한 내용은 342 페이지의 『참조를 해제하는 조회』에서 참조하십시오.

참조 무결성 및 범위 지정된 참조의 차이

비록 범위 지정된 참조가 테이블에 있는 오브젝트간의 관계를 정의해도 참조 무결성 관계와는 다릅니다. 범위는 목표 테이블에 대한 정보를 제공하기만 합니다. 그 정보는 목표 테이블에서 오브젝트를 참조 해제할 때 사용됩니다. 범위 지정된 참조는 다른 테이블에 있는 값을 요구하거나 강요하지 않습니다. 예를 들어, `Employee` 테이블에 있는 `Dept` 컬럼에는 `BusinessUnit` 테이블에 없는 `BusinessUnit` 오브젝트 식별자 컬럼에 대한 참조가 있습니다. 이런 관계의 오브젝트가 존재하려면 테이블간에 참조 무결성을 추가해야 합니다. 자세한 내용은 348 페이지의 『오브젝트 식별자 컬럼에 대한 제한조건 작성』에서 참조하십시오.

뷰 유형 작성

CREATE VIEW문을 사용하여 유형화 뷰를 작성할 수 있습니다. 예를 들어, 유형화 BusinessUnit 테이블의 뷰를 작성하려면, 원하는 속성이 있는 구조화 유형을 정의하고 그 유형을 사용하여 유형화 뷰를 작성합니다.

```
CREATE TYPE VBusinessUnit_t AS (Name VARCHAR(20))
   MODE DB2SQL;

CREATE VIEW VBusinessUnit OF VBusinessUnit_t MODE DB2SQL
   (REF IS VObjectID USER GENERATED)
   AS SELECT VBusinessUnit_t(VARCHAR(Oid)), Name FROM BusinessUnit;
```

CREATE VIEW문의 OF절은 DB2에게 표시된 구조화 유형의 속성을 뷰 컬럼의 기본값으로 사용하도록 지시합니다. 이 경우 DB2는 뷰의 컬럼의 기반으로 VBusinessUnit_t 구조화 유형을 사용합니다.

뷰의 VObjectID 컬럼에는 REF(VBusinessUnit_t)의 유형이 있습니다. REF(BusinessUnit_t)의 유형에서 REF(VBusinessUnit_t)의 유형으로 유형변환할 수 없으므로, 먼저 BusinessUnit 테이블에서 Oid 컬럼 값을 VARCHAR 데이터 유형으로 유형변환한 후 VARCHAR 데이터 유형에서 REF(VBusinessUnit_t) 데이터 유형으로 유형변환합니다.

MODE DB2SQL절은 유형화 뷰의 모드를 지정합니다. 이 모드는 현재 지원되는 유일한 유효 모드입니다.

REF IS...절은 입력된 CREATE TABLE문의 해당 절과 동일합니다. 이는 뷰의 첫번째 컬럼인 오브젝트 식별자 컬럼의 뷰(이 경우 VObjectID) 이름을 제공합니다. 루트 유형에서 유형화 뷰를 작성하면 뷰에 대해 오브젝트 식별자 컬럼을 지정해야 합니다. 부속 유형에서 유형화 뷰를 작성하면 뷰는 오브젝트 식별자 컬럼을 계승할 수 있습니다.

USER GENERATED절은 행을 삽입할 때 사용자가 오브젝트 식별자 컬럼의 초기 값을 제공해야 함을 지정합니다. 일단 삽입되면 오브젝트 식별자 컬럼을 갱신할 수 없습니다.

AS 키워드 다음에 오는 뷰의 본문은 뷰의 내용을 판별하는 SELECT문입니다. 이 SELECT문에서 리턴된 컬럼 유형은 초기 오브젝트 식별자 컬럼을 포함하여 유형화 뷰의 컬럼 유형과 호환될 수 있어야 합니다.

유형화 뷰 계층의 작성을 설명하기 위해 다음 예는 일부 민감 데이터를 생략하고, 이전에 329 페이지의 『유형화 테이블 작성』하에서 작성된 Person 테이블 계층에서 일부 유형 구별을 제거하는 뷰 계층를 정의합니다.

```
CREATE TYPE VPerson_t AS (Name VARCHAR(20))
   MODE DB2SQL;

CREATE TYPE VEmployee_t UNDER VPerson_t
   AS (Salary INT, Dept REF(VBusinessUnit_t))
   MODE DB2SQL;

CREATE VIEW VPerson OF VPerson_t MODE DB2SQL
   (REF IS VObjectID USER GENERATED)
   AS SELECT VPerson_t (VARCHAR(Oid)), Name FROM ONLY(Person);

CREATE VIEW VEmployee OF VEmployee_t MODE DB2SQL
   UNDER VPerson INHERIT SELECT PRIVILEGES
   (Dept WITH OPTIONS SCOPE VBusinessUnit)
   AS SELECT VEmployee_t(VARCHAR(Oid)), Name, Salary,
      VBusinessUnit_t(VARCHAR(Dept))
   FROM Employee;
```

두 CREATE TYPE문은 이 예의 오브젝트 뷰 계층을 작성하는 데 필요한 구조화 유형을 작성합니다.

위에서 첫번째 유형화 CREATE VIEW문은 계층의 루트 뷰인 VPerson을 작성하며, VBusinessUnit 뷰 정의와 매우 비슷합니다. 차이점은 ONLY(Person)을 사용하여 서브테이블이 아닌 Person 테이블에 있는 Person 테이블 계층의 행만 VPerson 뷰에 포함시킨다는 것입니다. 그러면 VPerson의 Oid 값이 VEmployee의 Oid 값에 비해 고유해집니다. 두 번째 CREATE VIEW문은 VPerson 뷰의 VEmployee 서브뷰를 작성합니다. CREATE TABLE...UNDER문에서의 UNDER 절과 같이 UNDER절은 뷰 계층을 설정합니다. 같은 스키마에서 해당 수퍼뷰로 서브뷰를 작성해야 합니다. 유형화 테이블처럼 서브뷰는 수퍼뷰에서 컬럼을 계승합니다. VEmployee 뷰에 있는 행은 VPerson에서 VObjectID와 Name 컬럼을 계승하고, VEmployee_t 유형에 연관된 추가 Salary와 Dept 컬럼이 있습니다.

INHERIT SELECT PRIVILEGES절은 유형화 REATE TABLE문을 발행할 때와 같이 CREATE VIEW문을 발행할 때 같은 효과를 가지고 있습니다. INHERIT SELECT PRIVILEGES절에 대한 자세한 내용은 331 페이지의 『SELECT 특권 계승 지정』를 참조하십시오. 유형화 뷰 정의에서의 WITH OPTIONS절은 유형화 테이블 정의에서와 같은 효과를 가집니다. WITH OPTIONS절을 사용하면 SCOPE와 같은 컬럼 옵션을 지정할 수 있습니다. READ ONLY절은 수퍼뷰 컬럼을 읽기 전용으로 강제 표시하여 히후에 서브뷰 정의가 읽기 전용인 같은 컬럼에 대한 표현식을 올바로 지정할 수 있게 합니다.

`VEmployee` 뷰의 `Dept` 컬럼처럼 뷰에 참조 컬럼이 있으면, SQL 참조 해제 조작에서 컬럼을 사용할 수 있도록 컬럼에 범위를 연관시켜야 합니다. 뷰의 참조 컬럼에 범위를 지정하지 않고 기본 테이블이나 뷰 컬럼에 범위가 지정되면 기본 컬럼의 범위는 뷰의 참조 컬럼으로 전달됩니다. WITH OPTION절을 사용하여 뷰의 참조 컬럼에 범위를 명시적으로 지정할 수 있습니다. 이전 예에서 `VEmployee` 뷰의 `Dept` 컬럼은 `VBusinessUnit` 뷰를 자체 범위로 수신합니다. 기본 테이블이나 뷰 컬럼에 범위가 없고 뷰 정의에 명시적으로 지정된 범위가 없거나 ALTER VIEW 문으로 지정된 범위가 없으면, 참조 컬럼은 범위가 지정되지 않은 상태로 남습니다.

유형화 뷰를 작성하고 사용하기 전에 유형화 뷰의 조회 제한과 연관하여 몇 가지 주요 규칙이 적용되는 *SQL 참조서*를 주의깊게 읽어보십시오.

사용자 정의 유형(UDT) 또는 유형 맵핑 제거

사용자는 DROP문을 사용하여 사용자 정의 유형(UDT) 또는 유형 맵핑을 제거할 수 있습니다. 유형 맵핑에 대해서는 647 페이지의 『데이터 유형 맵핑에 대한 작업』에서 참조하십시오. UDT를 다음과 같이 사용한 경우, UDT를 제거할 수 없습니다.

- 기존 테이블 또는 뷰의 컬럼 정의에서 사용된 경우.
- 기존 유형화 테이블 또는 유형화 뷰(구조화 유형)의 유형으로서 사용된 경우.
- 또다른 구조화된 유형의 상위 유형으로 사용된 경우.

기본 유형 맵핑을 제거할 수 없습니다. 또다른 맵핑을 작성하여 이를 대체만 할 수 있습니다.

데이터베이스 관리 프로그램은 이 UDT에 종속되어 있는 모든 사용자 정의 함수 (UDF)를 제거하려고 합니다. 뷰, 트리거, 테이블 점검 제한조건 또는 다른 UDF가 이에 종속적인 경우에 UDF가 제거될 수 없습니다. DB2가 종속 UDF를 제거하지 못하면 DB2는 UDT를 제거하지 않습니다. UDT를 제거하면 이것을 사용하는 다른 패키지나 캐쉬화된 동적 SQL문을 무효화합니다.

UDT에 대한 변환을 작성하고 그 UDT를 제거할 계획이라면, 연관 변환을 제거하는 것에 대해 고려해 보십시오. 변환을 제거하려면 DROP TRANSFORM문을 발행하십시오. DROP TRANSFORM문의 전체 구문은 *SQL 참조서*에 나와 있습니다. 사용자 정의 변환만 제거할 수 있습니다. 내장된 변환이나 연관된 그룹 정의를 제거할 수 없습니다.

뷰 변경 또는 제거

ALTER VIEW문은 참조 유형 컬럼이 범위를 추가하도록 변경하여 기존 뷰를 수정합니다. 뷰에 대한 다른 모든 변경을 수행할 때는 해당 뷰를 제거한 다음 다시 작성해야 합니다.

뷰를 변경할 때는 아직 범위가 정의되지 않은 기존 참조 유형 컬럼에 범위를 추가해야 합니다. 그리고 수퍼뷰로부터 컬럼을 계승할 수 없습니다.

ALTER VIEW문의 컬럼 이름 자료 유형은 REF(유형화 테이블 이름 또는 유형화 뷰 이름 유형)여야 합니다.

ALTER VIEW문에 대해서는 *SQL 참조서*에서 참조하십시오.

다음 예는 EMP_VIEW를 제거하는 방법을 보여줍니다.

```
DROP VIEW EMP_VIEW;
```

제거된 뷰에 종속되어 있는 모든 뷰는 작동 불능 상태가 됩니다. 작동 불능 뷰에 대한 자세한 내용은 *관리 안내서*의 "작동 불능 뷰 복구"절을 참조하십시오.

패키지 및 캐쉬화된 동적 명령문에 무효 표시가 되어 있더라도, 테이블 및 색인과 같은 기타 데이타베이스 오브젝트에는 영향을 주지 않습니다. 더 자세한 내용은 *관리 안내서*의 "명령문 종속성"절을 참조하십시오.

테이블 계층의 경우, 다음 예에서처럼 계층의 루트 뷰를 명명하여 한 명령문에서 전체 뷰 계층을 제거할 수 있습니다.

```
DROP VIEW HIERARCHY VPerson;
```

뷰 제거 및 작성에 대해서는 *SQL 참조서*에서 참조하십시오.

유형화 테이블 조회

필수 SELECT 권한이 있으면, 유형화되지 않은 테이블 조회와 같은 방법으로 유형화 테이블을 조회할 수 있습니다. 조회는 SELECT의 목표에서 행을 규정하여 요청된 컬럼과 그 모든 서브테이블을 리턴합니다. 예를 들어, Person 테이블 계층의 데이터에 대한 다음 조회는 모든 사람의 이름과 나이를 리턴합니다. 즉, Person 테이블과 그 서브테이블에 있는 모든 행입니다. 컬럼 중 하나가 구조화 유형 컬럼인 경우, 비슷한 조회의 작성에 대해서는 353 페이지의 『구조화 유형 값의 검색 및 수정』을 참조하십시오.

```
SELECT Name, Age
FROM Person;
```

조회 결과는 다음과 같습니다.

```
NAME                 AGE
-------------------- -----------
Andrew               29
Bob                  30
Cathy                25
Dennis               26
Eva                  31
Franky               28
Gordon               19
Helen                20
Iris                 35
Christina            10
Ken                  55
Leo                  35
Brian                 7
Susan                39
```

참조를 해제하는 조회

참조의 범위를 지정할 때마다, *참조 해제 조작*을 사용하여 외부 조인이나 상관 부속 조회가 필요할 수도 있는 조회를 발행합니다. Employee 테이블의 Dept 속성

과 BusinessUnit 테이블로 범위가 지정된 Employee의 서브테이블을 고려하십시오. 다음 예는 적절한 경우 데이터베이스에 있는 모든 직원의 이름, 급여 및 부서 이름 또는 널(NULL) 값을 리턴합니다. 즉, 조회는 Employee 테이블에 있는 모든 행과 Employee 서브테이블의 값을 리턴합니다. 상관 부속 조회나 외부 조인을 사용하여 비슷한 조회를 작성할 수 있습니다. 그러나, *참조 해제 연산자*(->)를 사용하여 Employee 테이블과 그 서브테이블에 있는 참조 컬럼에서 BusinessUnit 테이블로 경로를 따라가고, BusinessUnit 테이블의 Name 컬럼에서 결과를 리턴하는 것이 더 쉽습니다.

참조 해제 조작의 간단한 형식은 다음과 같습니다.

```
scoped-reference-expression -> column-in-target-typed-table
```

다음 조회는 참조 해제 연산자를 사용하여 BusinessUnit 테이블에서 Name 컬럼을 확보합니다.

```
SELECT Name, Salary, Dept->Name
   FROM Employee
```

조회 결과는 다음과 같습니다.

```
NAME                 SALARY      NAME
-------------------- ----------- --------------------
Dennis               30000       Toy
Eva                  45000       Shoe
Franky               39000       Shoe
Iris                 55000       Toy
Christina            85000       Toy
Ken                  105000      Shoe
Leo                  92000       Shoe
Brian                112000      Toy
Susan                37000.48    ---
```

자체 참조의 참조도 해제할 수가 있습니다. 337 페이지의 그림13에 정의되어 있는 부품 테이블을 고려하십시오. 다음 조회는 부품 공급자의 위치와 함께 날개에 직접 사용된 부품을 나열합니다.

```
SELECT P.Descript, P.Supplied_by ->Location
   FROM Parts P
   WHERE P.Used_in -> Descript='Wing';
```

DEREF 내장 함수

참조를 해제하여 DEREF 내장 함수를 통해 전체 구조화 오브젝트를 하나의 값으로 확보할 수 있습니다. 다음은 DEREF의 간단한 형식입니다.

```
DEREF (scoped-reference-expression)
```

DEREF는 보통 TYPE_NAME과 같은 다른 내장 함수의 컨텍스트에서 사용되거나 응용프로그램 외부로 바인딩하기 위해 전체 구조화 오브젝트를 확보합니다.

기타 유형-관련 내장 함수

DEREF 함수는 가끔 TYPE_NAME, TYPE_ID 또는 TYPE_SCHEMA 내장 함수의 일부로 호출됩니다. 이런 함수의 목적은 표현식의 동적 유형의 이름, 내부 ID 및 스키마 이름을 리턴합니다. 예를 들어, 다음 예에서는 Responsible이라는 속성의 Project 유형화 테이블을 작성합니다.

```
CREATE TYPE Project_t
   AS (Projid INT, Responsible REF(Employee_t))
   MODE DB2SQL;

CREATE TABLE Project
   OF Project_t (REF IS Oid USER GENERATED,
   Responsible WITH OPTIONS SCOPE Employee);
```

Responsible 속성은 Employee 테이블의 참조로 정의되므로, 직원뿐만 아니라 관리자와 아키텍트의 인스턴스를 참조할 수 있습니다. 응용프로그램에서 모든 행의 동적 유형의 이름을 알아야 되는 경우, 다음과 같은 조회를 사용할 수 있습니다.

```
SELECT Projid, Responsible->Name,
   TYPE_NAME(DEREF(Responsible))
   FROM PROJECT;
```

이전 예에서는 참조 해제 연산자를 사용하여 Employee 테이블에서 Name의 값을 리턴하고 DEREF 함수를 호출하여 Employee_t의 인스턴스에 대한 동적 유형을 리턴합니다.

이 절에서 설명된 내장 함수에 대한 자세한 내용은 *SQL 참조서*를 참조하십시오.

권한 부여 요구사항: DEREF 함수를 사용하려면, 테이블 계층에서 참조된 부분에 있는 모든 테이블과 서브테이블에 대해 SELECT 권한이 있어야 합니다. 예를 들어, 위의 조회에서 `Employee`, `Manager` 및 `Architect` 유형화 테이블에 대한 SELECT 특권이 있어야 합니다.

추가 조회 스펙 기술

ONLY를 사용하여 특정 유형의 오브젝트 리턴

조회에서 그 서브테이블을 제외한 특정 유형의 오브젝트만 리턴하려면 ONLY 키워드를 사용하십시오. 예를 들어, 다음 조회는 건축가 또는 관리자가 아닌 직원의 이름만 리턴합니다.

```
SELECT Name
FROM ONLY(Employee);
```

이전 조회는 다음 결과를 리턴합니다.

```
NAME
--------------------
Dennis
Eva
Franky
Susan
```

데이터의 보안을 보호하기 위해 ONLY를 사용할 때 `Employee`의 모든 서브테이블에서 SELECT 특권이 필요합니다.

ONLY절을 사용하여 UPDATE 또는 DELETE문의 조작을 명명된 테이블로 제한할 수도 있습니다. 즉, ONLY절은 조작이 그 명명된 테이블의 모든 서브테이블에서 발생하지 않게 합니다.

TYPE 술어를 사용하여 리턴된 유형 제한

좀더 일반적인 방법으로 SQL문에 의해 리턴되거나 영향을 받는 행을 제한하려면 Type 술어를 사용할 수 있습니다. Type 술어를 통해 표현식의 동적 유형을 하나 이상의 명명된 유형과 비교할 수 있습니다. Type 술어의 간단한 버전은 다음과 같습니다.

```
<expression> IS OF (<type_name>[, ...])
```

여기서 *expression*은 구조화 유형의 인스턴스를 리턴하는 SQL 표현식을 나타내고 *type_name*은 인스턴스를 비교하는 하나 이상의 구조화 유형을 나타냅니다.

예를 들어, 다음 조회는 35살 이상이고 관리자 또는 검축가인 사람을 리턴합니다.

```
SELECT Name
   FROM Employee E
   WHERE E.Age > 35 AND
   DEREF(E.Oid) IS OF (Manager_t, Architect_t);
```

이전 조회는 다음 결과를 리턴합니다.

```
NAME
--------------------
Ken
```

OUTER를 사용하여 가능한 모든 속성 리턴

DB2가 구조화 유형 행 값을 리턴하면, 응용프로그램은 특정 인스턴스에 포함되어 있거나 포함될 수 있는 속성을 모르고 있어도 됩니다. 예를 들어, 사람을 리턴하면, 그 사람은 사람의 속성만 가지고 있거나 직원, 관리자 또는 사람의 기타 부속 유형의 속성을 가지고 있을 수도 있습니다. 응용프로그램이 하나의 SQL 조회에서 가능한 모든 속성의 값을 확보해야 하는 경우, 테이블 참조에서 OUTER 키워드를 사용할 수 있습니다.

OUTER (*table-name*)와 OUTER(*view-name*)은 테이블의 컬럼이나 뷰로 구성된 가상 테이블과 그 서브테이블에서 소개된 추가 컬럼(있는 경우)을 리턴합니다. 추가 컬럼은 테이블 오른쪽에 추가되고 깊이의 순서로 서브테이블 계층을 따라갑니다. 공동 상위가 있는 서브테이블은 각각의 유형이 작성된 순서대로 통과됩니다. 행에는 *table-name*의 모든 행과 *table-name*의 서브테이블의 모든 추가 행이 들어 있습니다. 널 값은 행의 서브테이블에 없는 컬럼에 대해 리턴됩니다.

예를 들어, 표준 이상을 성취하려는 사람들에 대한 정보를 보고 싶을 때 OUTER를 사용할 수 있습니다. 다음 조회는 높은 급여 `Salary` 또는 높은 성적 평균 `GPA`가 이는 `Person` 테이블 계층에서 정보를 리턴합니다.

```
SELECT *
   FROM OUTER(Person) P
   WHERE P.Salary > 200000
   OR P.GPA > 3.95 ;
```

OUTER(Person)을 사용하면 Person 조회에서 가능하지 않은 부속 유형 속성을 참조할 수 있습니다.

OUTER를 사용하면 참조된 테이블의 모든 정보가 그 사용에 의해 노출되므로, 그 테이블의 모든 서브테이블 또는 뷰에 대한 SELECT 특권이 필요합니다.

응용프로그램에서 이렇게 성취욕이 강산 사람들의 속성뿐만 아니라, 각각에 대해 가장 고유한 유형이 무엇인지 알아야 한다고 가정해 보십시오. 오브젝트의 오브젝트 식별자를 TYPE_NAME 내장 함수로 전달하고, 이를 다음과 같이 OUTER 조회와 결합하여 이를 단일 조회에서 수행할 수 있습니다.

```
SELECT TYPE_NAME(DEREF(P.Oid)), P.*
   FROM OUTER(Person) P
   WHERE P.Salary > 200000 OR
   P.GPA > 3.95 ;
```

Person 유형화 테이블의 Address 컬럼에는 구조화 유형이 있으므로, 추가 함수를 정의하고 추가 SQL을 발행하여 그 컬럼에서 데이터를 리턴해야 합니다. 구조화 유형 컬럼에서 데이터를 리턴하는 것에 대한 자세한 내용은 353 페이지의 『구조화 유형 값의 검색 및 수정』을 참조하십시오. 추가 단계를 수행한다고 간주되면, 이전 조회는 *Additional Attributes*에 GPA와 Salary가 들어 있는 다음과 같은 출력을 리턴합니다.

```
1                  OID           NAME                 Additional Attributes
------------------ ------------- --------------------  ...
PERSON_T           a             Andrew                ...
PERSON_T           b             Bob                   ...
PERSON_T           c             Cathy                 ...
EMPLOYEE_T         d             Dennis                ...
EMPLOYEE_T         e             Eva                   ...
EMPLOYEE_T         f             Franky                ...
MANAGER_T          i             Iris                  ...
ARCHITECT_T        l             Leo                   ...
EMPLOYEE_T         s             Susan                 ...
```

도움말 및 추가 정보

시스템 생성 오브젝트 식별자 정의

DB2에서 고유 오브젝트 식별자를 자동으로 생성하려면 GENERATE_UNIQUE 함수를 사용합니다. GENERATE_UNIQUE는 CHAR (13) FOR BIT DATA 값

을 리턴하므로 CREATE TYPE문의 REF USING절은 그 유형의 값을 통합할 수 있습니다. VARCHAR (16) FOR BIT DATA의 기본값이 이 목적에 적합합니다. 예를 들어, BusinessUnit_t 유형이 기본 표현 유형과 함께 작성되었다고 가정하십시오. 즉, 다음과 같이 REF USING절이 지정되지 않았습니다.

```
CREATE TYPE BusinessUnit_t AS
   (Name VARCHAR(20),
   Headcount INT)
   MODE DB2SQL;
```

유형화 테이블 정의는 다음과 같습니다.

```
CREATE TABLE BusinessUnit OF BusinessUnit_t
(REF IS Oid USER GENERATED);
```

항상 USER GENERATED절을 제공해야 합니다.

행을 유형화 테이블에 삽입하는 INSERT문은 다음과 같을 수 있습니다.

```
INSERT INTO BusinessUnit (Oid, Name, Headcount)
   VALUES(BusinessUnit_t(GENERATE_UNIQUE( )), 'Toy' 15);
```

Toy 부서에 속하는 직원을 삽입하려면, 다음과 같이 부속 선택을 발행하여 BusinessUnit 테이블에서 오브젝트 식별자 컬럼의 값을 검색하고 값을 BusinessUnit_t 유형으로 유형변환하며, 그 값을 Dept 컬럼에 삽입하는 명령문을 사용할 수 있습니다.

```
INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary, Dept)
   VALUES(Employee_t('d'), 'Dennis', 26, 105, 30000,
     BusinessUnit_t(SELECT Oid FROM BusinessUnit WHERE Name='Toy'));
```

오브젝트 식별자 컬럼에 대한 제한조건 작성

오브젝트 식별자 컬럼을 외부 키의 상위 테이블의 키 컬럼으로 사용하려면, 먼저 유형화 테이블을 변경하여 명시적 고유 또는 기본 키 제한사항을 오브젝트 식별자 컬럼에 추가해야 합니다. 예를 들어, 349 페이지의 그림14에서와 같이 각 직원의 관리자가 항상 직원 테이블에 있는 직원으로 존재해야 하는 직원에 대한 자체 참조 관계를 작성한다고 가정하십시오.

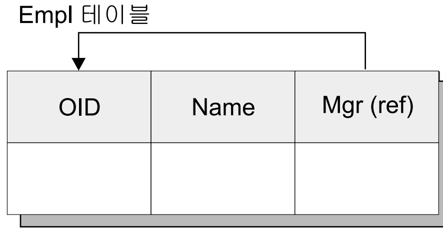


그림 14. 자체 참조 유형 예

자체 참조 관계를 작성하려면 다음 단계를 수행하십시오.

단계 1. 유형 작성

```
CREATE TYPE Empl_t AS
   (Name VARCHAR(10), Mgr REF(Empl_t))
   MODE DB2SQL;
```

단계 2. 유형화 테이블 작성

```
CREATE TABLE Empl OF Empl_t
(REF IS Oid USER GENERATED);
```

단계 3. 기본 또는 고유 제한조건을 Oid 컬럼에 추가하십시오.

```
ALTER TABLE Empl ADD CONSTRAINT pk1 UNIQUE(Oid);
```

단계 4. 외부 키 제한조건 추가

```
ALTER TABLE Empl ADD CONSTRAINT fk1 FOREIGN KEY(Mgr)
   REFERENCES Empl (Oid);
```

구조화 유형을 컬럼 유형으로 작성 및 사용

이 절에서는 사용자 정의 구조화 유형을 컬럼의 유형으로 사용하는 작업에 관련된 주요 타스크에 대해 설명합니다. 이 절을 읽기 전에 316 페이지의 『구조화 유형 개요』에 있는 재료에 익숙해 있어야 합니다.

컬럼에 구조화 유형 인스턴스 삽입

구조화 유형은 테이블, 뷰 또는 컬럼의 문맥에서 사용될 수 있습니다. 구조화 유형을 작성할 때 사용자 정의 유형 동작과 유형 속성을 둘다 캡슐화할 수 있습니다. 유형의 동작을 포함하려면 CREATE TYPE 또는 ALTER TYPE문으로 메소드 서명을 지정하십시오. 메소드 작성에 대한 자세한 내용은 409 페이지의 『제14장 사용자 정의 함수(UDFs)와 메소드』를 참조하십시오.

그림15는 이 절에서 예로 사용된 유형 계층을 보여줍니다. 루트 유형은 Address_t 인데 그 나라에서 주소를 구성하는 방법의 몇 가지 측면을 반영하는 추가 속성이 있는 3가지 부속 유형이 있습니다.

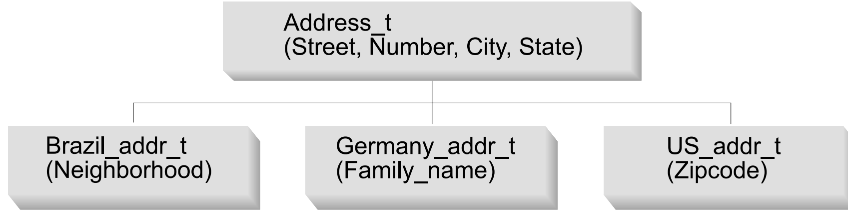


그림 15. Address_t 유형에 대한 구조화 유형 계층

```
CREATE TYPE Address_t AS
   (street VARCHAR(30),
   number CHAR(15),
   city VARCHAR(30),
   state VARCHAR(10))
   MODE DB2SQL;

CREATE TYPE Germany_addr_t UNDER Address_t AS
   (family_name VARCHAR(30))
   MODE DB2SQL;

CREATE TYPE Brazil_addr_t UNDER Address_t AS
   (neighborhood VARCHAR(30))
   MODE DB2SQL;

CREATE TYPE US_addr_t UNDER Address_t AS
   (zip CHAR(10))
   MODE DB2SQL;
```

구조화 유형 컬럼으로 테이블 정의

어떻게 구조화 유형을 데이터 레코드에 배치하는지에 대해 염려하지 않는 한, 구조화 유형의 컬럼으로 테이블을 작성하는 추가 구문이 없습니다. 예를 들어, 다음 명령문은 Address_t 유형의 컬럼을 Customer_List 유형화되지 않은 테이블에 추가합니다.

```
ALTER TABLE Customer_List
   ADD COLUMN Address Address_t;
```

이제 Address_t의 인스턴스 또는 Address_t의 부속 유형은 이 테이블에 저장될 수 있습니다. 구조화 유형 삽입에 대한 내용은 『구조화 유형 값이 있는 행 삽입』을 참조하십시오.

구조화 유형이 데이터 레코드에 배치되는 방법에 대해 걱정이 되면 CREATE TYPE문에 있는 INLINE LENGTH절을 사용하여 구조화 유형 컬럼의 인스턴스 최대 *크기*를 표시하여 값의 나머지와 함께 행에 인라인으로 저장할 수 있습니다. INLINE LENGTH절에 대한 자세한 정보는 *SQL 참조서*의 CREATE TYPE (Structured)문을 참조하십시오.

구조화 유형 속성으로 유형 정의

구조화 유형 속성으로 유형을 작성할 수 있지만 이를 변경(사용되기 전)하여 이런 속성을 추가하거나 제거할 수 있습니다. 예를 들어, 다음 CREATE TYPE문에는 Address_t 유형의 속성이 들어 있습니다.

```
CREATE TYPE Person_t AS
   (Name VARCHAR(20),
   Age INT,
   Address Address_t)
   REF USING VARCHAR(13)
   MODE DB2SQL;
```

Person_t는 테이블의 유형, 일반 테이블의 컬럼 유형 또는 다른 구조화 유형의 속성으로 사용될 수 있습니다.

구조화 유형 값이 있는 행 삽입

구조화 유형을 작성하면, DB2는 자동으로 그 유형의 구성자 메소드를 생성하고 유형의 속성에 대해 변환 및 관찰 메소드를 생성합니다. 이런 메소드를 사용하여 구조화 유형의 인스턴스를 작성하고 이 인스턴스를 테이블 컬럼에 삽입할 수 있습니다.

새 행을 Employee 유형화 테이블에 추가하고 그 행에 주소 를 포함시킨다고 가정하십시오. 내장 데이터 유형처럼 VALUES절과 함께 INSERT를 사용하여 이 행을 추가할 수 있습니다. 그러나, 값을 지정하여 주소에 삽입하면 시스템에서 제공하는 구성자 함수를 호출하여 값을 작성해야 합니다.

```
INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary, Dept, Address)
   VALUES(Employee_t('m'), 'Marie', 35, 005, 55000, BusinessUnit_t(2),
   US_addr_t ( ) 1
      ..street('Bakely Avenue') 2
      ..number('555') 3
      ..city('San Jose') 4
      ..state('CA') 5
      ..zip('95141')); 6
```

이전 명령문은 다음 타스크를 수행하여 US_addr_t 유형의 인스턴스를 작성합니다.

1. US_addr_t()에 대한 호출은 US_addr_t 유형의 구성자 함수를 호출하여 모든 속성 세트가 널(NULL) 값으로 설정된 유형의 인스턴스를 작성합니다.
2. ..street('Bakely Avenue')에 대한 호출은 street 속성에 대해 변환 메소드를 호출하여 그 값을 'Bakely Avenue'로 설정합니다.
3. ..number('555')에 대한 호출은 number 속성에 대한 변환 메소드를 호출하여 그 값을 '555'로 설정합니다.
4. ..city('San Jose')에 대한 호출은 city 속성에 대한 변환 메소드를 호출하여 그 값을 'San Jose'로 설정합니다.
5. ..state('CA')에 대한 호출은 state 속성을 호출하여 그 값을 'CA'로 설정합니다.
6. ..zip('95141')에 대한 호출은 zip 속성을 호출하여 그 값을 '95141'로 설정합니다.

비록 Employee 테이블에 있는 Address 컬럼의 유형이 Address_t 유형으로 정의되어 있어도 대체 가능성의 특성은 US_addr_t는 Address_t의 부속 유형이므로, 여기에 US_addr_t의 인스턴스를 상주시킬 수 있음을 의미합니다.

유형의 인스턴스를 작성할 때마다 구조화 유형의 각 속성에 대한 변환 메소드를 명시적으로 호출하지 않도록 모든 속성을 초기화하는 SQL 본문 구성자 함수를 정의하십시오. 다음 예에는 US_addr_t 유형에 대한 SQL 본문 구성자 함수의 선언이 들어 있습니다.

```
CREATE FUNCTION US_addr_t
      (street Varchar(30),
      number Char(15),
      city Varchar(30),
      state Varchar(20),
      zip Char(10))
```

```
RETURNS US_addr_t
LANGUAGE SQL
RETURN Address_t()..street(street)..number(number)
    ..city(city)..state(state)..zip(zip);
```

다음 예는 이전 예에서 SQL 본문 구성자 함수를 호출하여 US_addr_t 유형의 인스턴스를 작성하는 방법을 보여줍니다.

```
INSERT INTO Employee(Oid, Name, Age, SerialNum, Salary, Dept, Address)
   VALUES(Employee_t('m'), 'Marie', 35, 005, 55000, BusinessUnit_t(2),
      US_addr_t('Bakely Avenue', '555', 'San Jose', 'CA', '95141'));
```

구조화 유형 값의 검색 및 수정

구조화 유형 컬럼에서 응용프로그램과 사용자 정의 함수가 데이터를 액세스할 수 있는 여러 가지 방법이 있습니다. 오브젝트를 단일 값으로 취급하려면, 먼저 359 페이지의 『호스트 언어 프로그램에 대한 맵핑 작성: 변환 함수』에 설명되어 있는 *변환 함수*를 정의해야 합니다. 일단 올바른 변환 함수를 정의하면 다른 값에서 선택할 수 있는 것만큼 구조화 오브젝트를 선택할 수 있습니다.

```
SELECT Name, Dept, Address
   FROM Employee
   WHERE Salary > 20000;
```

그러나, 이 절에서 DB2 내장 관찰 및 변환 메소드를 호출하여 오브젝트의 개별적인 속성을 명시적으로 액세스하는 방법을 설명합니다. 내장 메소드에서는 변환 함수를 정의하지 않아도 됩니다.

검색 속성

오브젝트의 개별적인 속성을 명시적으로 액세스하려면 이런 속성에 대해 DB2 내장 *관찰* 메소드를 호출하십시오. 관찰 메소드를 사용하면 오브젝트를 단일 값으로 취급하는 대신 속성을 개별적으로 검색할 수 있습니다.

다음 예는 `Address` 컬럼에 대해 정의된 정적 유형인 `Address_t`에서 관찰 메소드를 호출하여 `Address` 컬럼에 있는 데이터를 액세스할 수 있습니다.

```
SELECT Name, Dept, Address..street, Address..number, Address..city,
   Address..state
   FROM Employee
   WHERE Salary > 20000;
```

주: DB2는 *<type-name>..<method-name>*() 또는 *<type-name>..<method-name>*을 사용하여 매개변수를 취하지 않는 메소드를 호출할 수 있게 하는데, 여기서 *type-name*은 구조화 유형의 이름을 나타내고 *attribute-name*은 매개변수를 취하지 않는 메소드의 이름을 나타냅니다.

관찰 메소드를 사용하여 다음과 같이 호스트 변수에서 각 속성을 선택할 수도 있습니다.

```
SELECT Name, Dept, Address..street, Address..number, Address..city,
   Address..state
   INTO :name, :dept, :street, :number, :city, :state
   FROM Employee
   WHERE Empno = '000250';
```

부속 유형의 속성 액세스

Employee 테이블에서 주소는 Address_t, US_addr_t, Brazil_addr_t 그리고 Germany_addr_t의 4 가지 유형으로 되어 있습니다. 이전 예는 Address_t 정적 유형의 속성만 액세스합니다. Address_t의 부속 유형 중 하나에서 값의 속성을 액세스하려면 TREAT 표현식을 *사용하여* DB2에게 특정 오브젝트가 US_addr_t, Germany_addr_t 또는 Brazil_addr_t 유형이 될 수 있음을 알려줄 수 있습니다. TREAT 표현식은 구조화 유형 표현식을 다음 조회에서와 같이 그 부속 유형 중 하나로 유형변환합니다.

```
SELECT Name, Dept, Address..street, Address..number, Address..city,
   Address..state,
   CASE
      WHEN Address IS OF (US_addr_t)
      THEN TREAT(Address AS US_addr_t)..zip
      WHEN Address IS OF (Germany_addr_t)
      THEN TREAT (Address AS Germany_addr_t)..family_name
      WHEN Address IS OF (Brazil_addr_t)
      THEN TREAT (Address AS Brazil_addr_t)..neighborhood
   ELSE NULL END
   FROM Employee
   WHERE Salary > 20000;
```

주: 이전 접근 방법만 사용하여 부속 유형의 속성이 모두 같은 유형이거나 같은 유형으로 유형변환할 수 있을 때 구조화 유형의 부속 유형을 결정할 수 있습니다. 이전 예에서 zip, family_name, neighborhood는 모든 VARCHAR 또는 CHAR 유형이고 같은 유형으로 유형변환할 수 있습니다.

TREAT 표현식이나 TYPE 술어의 구문에 대한 자세한 내용은 *SQL 참조서*를 참조하십시오.

속성 수정

구조화 컬럼 값의 속성을 변경하려면 변경할 속성에 대해 변환 메소드를 호출하십시오. 예를 들어, 주소의 street 속성을 변경하려면 street에 대해 변경될 값을 가지고 변환 메소드를 호출할 수 있습니다. 리턴 값은 street의 새 값이 들어 있는 주소입니다. 다음 예는 street라는 속성에 대해 변환 메소드를 호출하여 Employee 테이블에서 주소 유형을 갱신합니다.

```
UPDATE Employee
   SET Address = Address..street('Bailey')
   WHERE Address..street = 'Bakely';
```

다음 예는 이전 예와 같은 갱신을 수행하지만 갱신에 대해 구조화 컬럼을 명명하는 대신 SET절은 street라는 속성에 대해 변환 메소드를 직접 액세스합니다.

```
UPDATE Employee
   SET Address..street = 'Bailey'
   WHERE Address..street = 'Bakely';
```

유형에 대한 정보 리턴

344 페이지의『기타 유형-관련 내장 함수』에 설명된 대로 내장 함수를 사용하여 특정 유형의 이름, 스키마 또는 내부 유형 ID를 리턴할 수 있습니다. 다음 명령문은 'Iris'라고 하는 직원에 연관된 주소의 정확한 유형을 리턴합니다.

```
SELECT TYPE_NAME(Address)
   FROM Employee
   WHERE Name='Iris';
```

유형과 변환 연관

변환 함수는 보통 쌍으로 일어납니다. 하나는 FROM SQL 변환 함수이고 다른 하나는 TO SQL 변환 함수입니다. FROM SQL 함수는 구조화 유형 오브젝트를 외부 프로그램에서 실행될 수 있는 유형으로 변환하고 TO SQL 함수를 오브젝트를 구성합니다. 변환 함수를 작성할 때 변환 함수의 각 논리적 쌍을 그룹에 넣습니다. *변환 그룹* 이름은 주어진 구조화 유형에 대해 이런 함수의 쌍을 식별합니다.

변환 함수를 사용하기 전에 CREATE TRANSFORM문을 사용하여 변환 함수를 그룹 이름 및 유형에 연관시켜야 합니다. CREATE TRANSFORM문은 하나 이

상의 기존 함수를 식별하고 이를 변환 함수로 사용합니다. 다음 예는 Address_t 유형에 대해 변환 함수로 사용될 함수의 두 쌍을 명명합니다. 명령문은 func_group과 client_group의 두 변환 그룹을 작성하는 데, 각각 FROM SQL 변환과 TO SQL 변환으로 구성됩니다.

```
CREATE TRANSFORM FOR Address_t
   func_group ( FROM SQL WITH FUNCTION addresstofunc,
      TO SQL WITH FUNCTION functoaddress )
   client_group ( FROM SQL WITH FUNCTION stream_to_client,
      TO SQL WITH FUNCTION stream_from_client ) ;
```

CREATE TRANSFORM문에 더 많은 그룹을 추가하여 추가 함수를 Address_t에 연관시킬 수 있습니다. 변환 정의를 변경하려면 CREATE TRANSFORM문을 추가 함수와 함께 재발행해야 합니다. 여러 호스트 언어 프로그램에 대해 클라이언트 함수를 사용자 정의할 수 있습니다. 예를 들어, C에 대해 하나 그리고 Java에 대해 하나 사용자 정의합니다. 응용프로그램의 성능을 최적화하려면 오브젝트 속성의 부속 집합에 대해서만 변환이 적용되어야 합니다. 또는 VARCHAR를 오브젝트의 클라이언트 표현으로 사용하는 변환 하나와 BLOB를 사용하는 변환 하나를 원할 수 있습니다.

SQL문 DROP TRANSFORM을 사용하여 유형에서 변환 함수의 연관을 해제시키십시오. DROP TRANSFORM문을 실행한 다음 함수는 계속 존재하지만 더 이상 이 유형에 대해 변환 함수로 사용되지 않습니다. 다음 예는 Address_t 유형에 대해 func_group 변환 함수의 특정 그룹을 연관해제하고, Address_t 유형에 대해 모든 변환 함수를 연관해제합니다.

```
DROP TRANSFORMS func_group FOR Address_t;

DROP TRANSFORMS ALL FOR Address_t;
```

변환 그룹 명명에 대한 권장사항

변환 그룹 이름은 *규정되지 않은* 식별자입니다. 즉, 이는 특정 스키마에 연관되어 있지 않습니다. 372 페이지의 『DB2에서 부속 유형 데이터 검색(바인드아웃)』에 설명된 대로 부속 유형 매개변수를 처리하기 위해 변환을 작성하지 않는 한, 모든 구조화 유형에 다른 변환 그룹 이름을 지정하지 말아야 합니다. 같은 프로그램 또는 같은 SQL문에서 관련되지 않은 여러 유형을 사용해야 하므로, 변환 그룹을 변환 함수에서 수행되는 타스크에 따라 명명해야 합니다.

변환 그룹의 이름은 일반적으로 유형 이름에 의존하지 않고 또는 수행하는 함수를 반영하거나 변환 함수의 논리(서로 다른 유형간에는 매우 다를 수 있음)를 반영해야 합니다. 예를 들어, TO와 FROM SQL 함수 변환이 정의되는 그룹에 대해 func_group 또는 object_functions 이름을 사용할 수 있습니다. TO와 FROM SQL 클라이언트 변환이 들어 있는 그룹에 대해 client_group 또는 program_group 이름을 사용할 수 있습니다.

다음 예에서 Address_t와 Polygon 유형은 매우 다른 변환을 사용하지만 같은 함수 그룹 이름을 사용합니다.

```
CREATE TRANSFORM FOR Address_t
   func_group (TO SQL WITH FUNCTION functoaddress,
   FROM SQL WITH FUNCTION addresstofunc );

CREATE TRANSFORM FOR Polygon
   func_group (TO SQL WITH FUNCTION functopolygon,
   FROM SQL WITH FUNCTION polygontofunc);
```

일단 358 페이지의 『변환 그룹 지정 위치』의 설명대로 적합한 상황에서 변환 그룹을 func_group으로 설정하면 DB2는 주소 또는 다각형을 바인드인하거나 바인드아웃할 때마다 올바른 변환 함수를 호출합니다.

제한사항: 'SYS' 문자열과 함께 변환 그룹을 시작하지 마십시오. 이 그룹은 DB2 용으로 예약되어 있습니다.

외부 함수나 메소드를 정의하고 변환 그룹 이름을 지정하지 않으면 DB2는 DB2_FUNCTION 이름을 사용하고 그 그룹 이름이 주어진 구조화 유형에 대해 지정되었다고 간주합니다. 주어진 구조화 유형을 참조하는 클라이언트 프로그램을 사전 처리 컴파일할 때 그룹 이름을 지정하지 않으면 DB2는 DB2_PROGRAM 이라고 하는 그룹 이름을 사용하고 다시 그룹 이름이 그 유형에 대해 정의되었다고 간주합니다.

이 기본 동작은 몇 경우 편리하지만 좀 더 복잡한 데이터베이스 스키마에서는 변환 그룹 이름에 대해 약간 더 광범위한 규칙이 필요합니다. 예를 들어, 유형을 바인드아웃할 다른 언어에 다른 그룹 이름을 사용하는 데 도움을 줍니다.

변환 그룹 지정 위치

주어진 구조화 유형에 대해 정의된 변환 그룹이 많을 때 프로그램 또는 특정 SQL 문에서 그 유형에 사용할 변환 그룹을 지정해야 합니다. 다음과 같이 3가지 경우에 변환 그룹을 지정해야 합니다.

- 외부 함수 또는 메소드가 정의되는 경우 참조된 오브젝트를 *분해*하고 *구성*하는 그룹을 지정해야 합니다. 자세한 내용은 『외부 루틴에 대해 변환 그룹 지정』의 내용을 참조하십시오.
- 정적 SQL을 사전 처리 컴파일하거나 바인드할 때 참조된 유형에 대해 클라이언트 바이드인과 바인드아웃을 수행하는 변환 그룹을 지정해야 합니다. 자세한 내용은 359 페이지의 『정적 SQL에 대한 변환 그룹 설정』의 내용을 참조하십시오.
- 동적 SQL을 실행하거나 명령행 프로세서를 사용할 때 참조된 유형에 대해 클라이언트 바인드인와 바인드아웃을 수행하는 변환 그룹을 지정해야 합니다. 자세한 내용은 359 페이지의 『동적 SQL에 대한 변환 그룹 설정』의 내용을 참조하십시오.

외부 루틴에 대해 변환 그룹 지정

CREATE FUNCTION과 CREATE METHOD문은 LANGUAGE절의 값이 SQL이 아닐 때만 유효한 TRANSFORM GROUP절을 지정할 수 있게 합니다. SQL 언어 함수에서는 변환이 필요없지만 외부 함수에서는 변환이 필요합니다. TRANSFORM GROUP절을 통해 주어진 함수 또는 메소드에 대해 구조화 유형 매개변수 및 결과에 사용되는 TO SQL과 FROM SQL 변환이 들어 있는 변환 그룹을 지정할 수 있습니다. 다음 예에서 CREATE FUNCTION과 CREATE METHOD문은 TO SQL과 FROM SQL 변환에 대해 `func_group` 변환 그룹을 지정합니다.

```
CREATE FUNCTION stream_from_client (VARCHAR (150))
RETURNS Address_t
   ...
   TRANSFORM GROUP func_group
   EXTERNAL NAME 'addressudf!address_stream_from_client'
   ...

CREATE METHOD distance ( point )
```

```
FOR polygon
RETURNS integer
:
TRANSFORM GROUP func_group ;
```

동적 SQL에 대한 변환 그룹 설정

동적 SQL을 사용하면 CURRENT DEFAULT TRANSFORM GROUP 특수 레지스터를 설정할 수 있습니다. 이 특수 레지스터는 정적 SQL문이나 외부 함수 또는 메소드와의 매개변수 및 결과 교환에 사용되지 않습니다. SET CURRENT DEFAULT TRANSFORM GROUP문을 사용하여 동적 SQL문에 대해 기본 변환 그룹을 설정하십시오.

```
SET CURRENT DEFAULT TRANSFORM GROUP = client_group;
```

정적 SQL에 대한 변환 그룹 설정

정적 SQL에 대해 PRECOMPILE 또는 BIND 명령에서 TRANSFORM GROUP 옵션을 사용하여 정적 SQL문에서 사용되는 정적 변환 그룹을 지정함으로써 다양한 유형의 값을 호스트 프로그램과 교환하십시오. 정적 변환 그룹은 동적 SQL문에 적용되지 않거나 외부 함수 또는 메소드와의 매개변수 및 결과 교환에 적용되지 않습니다. PRECOMPILE 또는 BIND 명령에서 정적 변환 그룹을 지정하려면 TRANSFORM GROUP절을 사용하십시오.

```
PRECOMPILE ...
TRANSFORM GROUP client_group
... ;
```

PRECOMPILE 및 BIND 명령에 대한 자세한 내용은 *Command Reference*를 참조하십시오.

호스트 언어 프로그램에 대한 맵핑 작성: 변환 함수

비록 353 페이지의 『검색 속성』에 설명된 대로 응용프로그램은 직접 전체 오브젝트를 선택할 수 없어도 오브젝트의 개별적인 속성으로 응용프로그램에 선택할 수 있습니다. 응용프로그램은 구성자 함수의 호출 결과를 삽입할 수 있어도 보통 전체 오브젝트를 직접 삽입하지 않습니다.

```
INSERT INTO Employee(Address) VALUES (Address_t());
```

서버와 클라이언트 응용프로그램간에 전체 오브젝트를 교환하려면 *변환 함수*를 작성해야 합니다.

변환 함수는 DB2가 오브젝트를 그 내용을 액세스하거나 오브젝트를 *바인드아웃*하는 잘 정의된 형식으로 변환하는 방법을 정의합니다. 다른 변환 함수는 DB2가 오브젝트를 데이터베이스에 저장하도록 리턴하거나 오브젝트를 *바인드인*하는 방법을 정의합니다. 오브젝트를 바인드아웃하는 변환은 FROM SQL 변환 함수라고 하고 컬럼을 바인드인하는 변환은 TO SQL 변환이라고 합니다.

대부분의 경우 오브젝트를 클라이언트 응용프로그램으로 전달하는 것보가 *루틴*이나 외부 UDF와 메소드로 전달하는 변환이 여러 가지 있습니다. 오브젝트를 외부 루틴으로 전달하면 오브젝트를 *분해*하여 루틴을 매개변수 목록으로 전달하기 때문입니다. 클라이언트 응용프로그램을 사용하면 오브젝트를 BLOB와 같은 단일 내장 유형을 변경할 수 있습니다. 이 프로세스를 오브젝트 인코딩이라고 합니다. 이런 두 유형의 변환은 함께 사용됩니다.

SQL문 CREATE TRANSFORM을 사용하여 변환 함수를 특정 *구조화* 유형과 연관시킬 수 있습니다. CREATE TRANSFORM문에서 함수는 *변환 그룹* 으로 그룹지어질 수 있습니다. 특정 변환용으로 사용되는 함수를 더 쉽게 식별할 수 있습니다. 각 변환 그룹에서 특정 유형에 대해 FROM SQL 변환과 TO SQL 변환은 하나 이상 될 수 없습니다.

주: 다음 주제는 응용프로그램이 항상 Address_t와 같이 알려진 정확한 유형을 수신하는 간단한 경우를 설명합니다. 이런 주제는 외부 루틴 또는 클라이언트 프로그램이 Address_t, Brazil_addr_t, Germany_addr_t 또는 US_addr_t를 수신하는 시나리오를 설명하지 않습니다. 그러나, 기본 프로세스를 좀 더 복잡한 경우에 적용하기 전에 기본 프로세스를 이해해야 합니다. 이렇게 복잡한 경우는 외부 루틴이나 클라이언트가 유형 또는 그 부속 유형을 동적으로 처리해야 하는 경우입니다. 부속 유형 인스턴스를 동적으로 처리하는 방법에 대한 자세한 내용은 372 페이지의『DB2에서 부속 유형 데이터 검색(바인드아웃)』을 참조하십시오.

외부 루틴과의 오브젝트 변경: 함수 변환

이 절에서는 *함수 변환*이라는 변환의 특정 유형을 설명합니다. DB2는 이런 TO SQL과 FROM SQL 함수 변환을 사용하여 오브젝트를 외부 루틴과 상호 전달합니다. SQL 본문 루틴에서는 변환을 사용할 필요가 없습니다. 그러나, 367 페이지의 『프로그램과의 오브젝트 교환: 클라이언트 변환』에 설명된 대로 DB2는 이런 함수를 클라이언트 프로그램과 오브젝트를 상호 전달하는 프로세스의 일부로 사용합니다.

다음 예는 주소를 입력 매개변수로 취하고 주소를 수정(예를 들어, 거리 이름의 변경을 반영하기 위해)하고 수정된 주소를 리턴하는 MYUDF라는 외부 UDF를 호출합니다.

```
SELECT MYUDF(Address)
FROM PERSON;
```

362 페이지의 그림16은 DB2가 주소를 처리하는 방법을 보여줍니다.

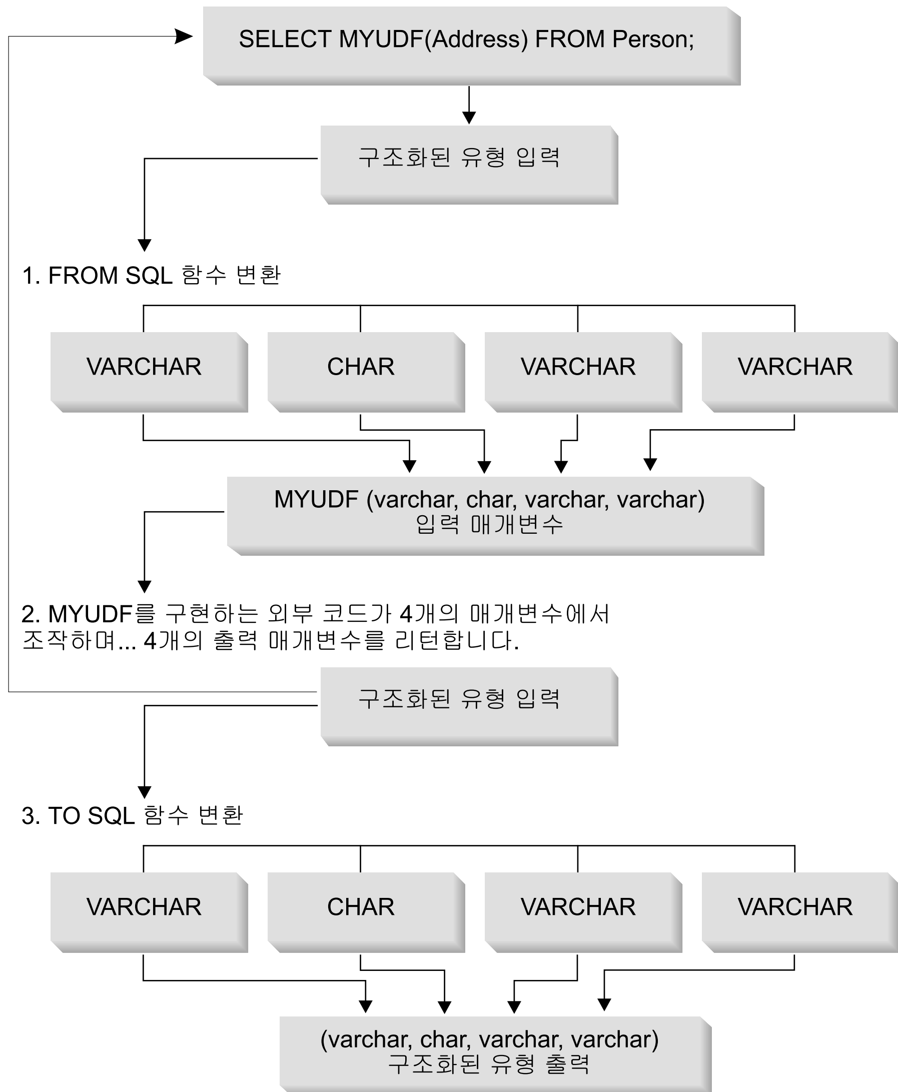


그림 16. 외부 루틴과의 구조화 유형 매개변수 교환

1. FROM SQL 변환 함수는 구조화 오브젝트를 순서가 정해진 기본 속성 세트로 분해합니다. 이를 통해 루틴은 오브젝트를 기본 내장 데이터 유형인 간단한 매개변수 목록으로 수신합니다. 예를 들어, 주소 오브젝트를 외부 루틴에 전달한다고 가정하십시오. `Address_t`의 속성은 VARCHAR, CHAR, VARCHAR 및 VARCHAR이며 이 순서를 유지합니다. 이 오브젝트를 루틴에 전달하는 FROM SQL 변환은 이 오브젝트를 입력으로 승인하고 VARCHAR, CHAR, VARCHAR 및 VARCHAR를 리턴해야 합

니다. 이런 출력은 4개의 해당 널 표시기 매개변수와 구조화 유형 자체에 대한 널 표시기가 있는 별도 매개변수 4개로 외부 루틴에 전달됩니다. Address_t 유형을 리턴하는 함수가 같은 순서를 사용하는 한, FROM SQL 함수에서 매개변수의 순서는 중요하지 않습니다. 자세한 내용은 365 페이지의 『외부 루틴에 구조화 유형 매개변수 전달』의 내용을 참조하십시오.

2. 외부 루틴은 분해된 주소를 입력 매개변수로 승인하고 이런 값을 처리한 후 속성을 출력 매개변수로 리턴합니다.
3. TO SQL 변환 함수는 MYUDF에서 리턴된 VARCHAR, CHAR, VARCHAR 및 VARCHAR 매개변수를 다시 Address_t 유형의 오브젝트로 변환해야 합니다. 다시 말해서 TO SQL 함수는 4개의 매개변수와 해당 널(NULL) 표시기 매개변수를 루틴에서 출력 매개변수로 취해야 합니다. TO SQL 함수는 구조화 오브젝트를 구성한 후 속성을 주어진 값으로 변환합니다.

주: MYUDF가 구조화 유형을 리턴하면, 다른 변환 함수는 UDF가 SELECT절에서 사용되는 경우 결과 구조화 유형을 변환해야 합니다. 다른 변환 함수를 작성하지 않으려면 다음 예에서와 같이 관찰 메소드에서 SELECT문을 사용할 수 있습니다.

```
SELECT Name
   FROM Employee
   WHERE MYUDF(Address)..city LIKE 'Tor%';
```

SQL 본문 루틴을 사용한 함수 변환 구현: 외부 루틴과 오브젝트를 교환할 때 오브젝트를 분해하고 구성하려면, SQL로 작성된 SQL 본문 루틴이라는 사용자 정의 함수를 사용할 수 있습니다. SQL 본문 루틴을 작성하려면, LANGUAGE SQL 절과 함께 CREATE FUNCTION문을 발행하십시오.

SQL 본문 함수에서 구성자, 관찰 및 변환을 사용하여 변환할 수 있습니다. 362 페이지의 그림16에서와 같이 이 SQL 본문 변환은 SQL문과 외부 함수 사이에 끼어듭니다. FROM SQL 변환은 오브젝트를 SQL 매개변수로 취하고 구조화 유형의 속성을 나타내는 값의 행을 리턴합니다. 다음 예에는 SQL 본문 함수를 사용하는 주소 오브젝트의 FROM SQL 변환 함수가 있습니다.

```
CREATE FUNCTION addresstofunc (A Address_t)  1
   RETURNS ROW  (Street VARCHAR(30), Number CHAR(15),
      City VARCHAR(30), State (VARCHAR(10))  2

   LANGUAGE SQL  3
   RETURN VALUES (A..Street, A..Number, A..City, A..State)  4
```

다음 목록은 이전 CREATE FUNCTION문의 구문을 설명합니다.

1. 이 함수의 서명은 매개변수 하나와 Address_t 유형의 오브젝트를 승인함을 나타냅니다.
2. RETURNS ROW절은 함수가 Street, Number, City State의 4개의 컬럼으로 구성된 행을 리턴함을 나타냅니다.
3. LANGUAGE SQL절은 이것이 외부 루틴이 아닌 SQL 본문 함수임을 나타냅니다.
4. RETURN절은 함수 본문의 시작을 표시합니다. 본문은 Address_t 오브젝트의 각 속성에 대해 관찰 메소드를 호출하는 단일 VALUES절로 구성됩니다. 관찰 메소드는 오브젝트를 함수가 행으로 리턴하는 기본 유형 세트로 분해합니다.

DB2는 사용자가 이 함수를 변환 함수로 사용하려고 한다는 것을 모르고 있습니다. 사용자가 이 함수를 사용하는 변환 그룹으로 작성하고, 그 변환 그룹을 적합한 상황으로 지정할 때까지 DB2는 함수를 변환 함수로 사용할 수 없습니다. 자세한 내용은 355 페이지의 『유형과 변환 연관』의 내용을 참조하십시오.

TO SQL 변환은 FROM SQL 함수와 역으로 작동합니다. 이는 루틴에서 매개변수 목록을 입력으로 취하고 구조화 유형의 인스턴스를 리턴합니다. 오브젝트를 구성하기 위해 다음 FROM SQL 함수는 Address_t 유형에 대해 구성자 함수를 호출합니다.

```
CREATE FUNCTION functoaddress (street VARCHAR(30), number CHAR(15),
                              city VARCHAR(30), state VARCHAR(10))  1
   RETURNS Address_t  2
   LANGUAGE SQL
   CONTAINS SQL
   RETURN
      Address_t()..street(street)..number(number)
      ..city(city)..state(state)  3
```

다음 목록은 이전 명령문의 구문을 설명합니다.

1. 함수는 기본 유형 속성 세트를 취합니다.
2. 함수는 Address_t 구조화 유형을 리턴합니다.
3. 함수는 Address_t에 대해 구성자를 호출하고, 각 속성에 대해 변환을 호출하여 입력 유형에서 오브젝트를 구성합니다.

주소를 리턴하는 모든 함수가 같은 순서를 사용하는 한 FROM SQL 함수에서 매개변수의 순서는 중요하지 않습니다.

외부 루틴에 구조화 유형 매개변수 전달: 구조화 유형 매개변수를 외부 루틴에 전달할 때 각 속성에 대해 매개변수를 전달해야 합니다. 각 매개변수에 널 표시기를 전달하고 구조화 유형 자체에 널(NULL) 표시기를 전달해야 합니다. 다음 예는 Address_t 구조화 유형을 승인하고 기본 유형을 리턴합니다.

```
CREATE FUNCTION stream_to_client (Address_t)
   RETURNS VARCHAR(150) ...
```

외부 루틴은 Address_t 유형(address_ind)의 인스턴스에 대해 널(NULL) 표시기를 승인하고, Address_t 유형의 각 속성에 대해 널(NULL) 표시기를 하나 승인해야 합니다. VARCHAR 출력 매개변수에 대해서도 널(NULL) 표시기가 있습니다. 다음 코드는 UDF를 구현하는 C 언어 함수 헤더를 나타냅니다.

```
void SQL_API_FN stream_to_client(
/*decomposed address*/
   SQLUDF_VARCHAR *street,
   SQLUDF_CHAR *number,
   SQLUDF_VARCHAR *city,
   SQLUDF_VARCHAR *state,
   SQLUDF_VARCHAR *output,
/*null indicators for type attributes*/
   SQLUDF_NULLIND *street_ind,
   SQLUDF_NULLIND *number_ind,
   SQLUDF_NULLIND *city_ind,
   SQLUDF_NULLIND *state_ind,
/*null indicator for instance of the type*/
   SQLUDF_NULLIND *address_ind,
/*null indicator for the VARCHAR output*/
   SQLUDF_NULLIND *out_ind,
   SQLUDF_TRAIL_ARGS)
```

외부 루틴으로 구조화 유형 매개변수 전달: 복잡: 루틴이 서로 다른 두 구조화 유형 매개변수 *st1*과 *st2*를 승인하고, *st3*의 다른 구조화 유형을 리턴한다고 가정하십시오.

```
CREATE FUNCTION myudf (int, st1, st2)
   RETURNS st3
```

표 14. myudf 매개변수 속성

ST1	ST2	ST3
st1_att1 VARCHAR	st2_att1 VARCHAR	st3_att1 INTEGER
st2_att2 INTEGER	st2_att2 CHAR	st3_att2 CLOB
	st2_att3 INTEGER	

다음 코드는 UDF를 구현하는 C 언어 헤더 루틴을 나타냅니다. 인수에는 다음과 같이 분해된 구조화 유형의 속성에 대한 변수와 널(NULL) 표시기 및 구조화 유형의 각 인스턴스에 대한 널(NULL) 표시기가 있습니다.

```
void SQL_API_FN myudf(
     SQLUDF_INTEGER *INT,
  /* Decompose st1 input */
     SQLUDF_VARCHAR *st1_att1,
     SQLUDF_INTEGER  *st1_att2,
  /* Decompose st2 input */
     SQLUDF_VARCHAR *st2_att1,
     SQLUDF_CHAR   *st2_att2,
     SQLUDF_INTEGER *st2_att3,
  /* Decompose st3 output */
     SQLUDF_VARCHAR *st3_att1out,
     SQLUDF_CLOB   *st3_att2out,
  /* Null indicator of integer*/
     SQLUDF_NULLIND *INT_ind,
  /* Null indicators of st1 attributes and type*/
     SQLUDF_NULLIND *st1_att1_ind,
     SQLUDF_NULLIND *st1_att2_ind,
     SQLUDF_NULLIND *st1_ind,
  /* Null indicators of st2 attributes and type*/
     SQLUDF_NULLIND *st2_att1_ind,
     SQLUDF_NULLIND *st2_att2_ind,
     SQLUDF_NULLIND *st2_att3_ind,
     SQLUDF_NULLIND *st2_ind,
  /* Null indicators of st3_out attributes and type*/
     SQLUDF_NULLIND *st3_att1_ind,
     SQLUDF_NULLIND *st3_att2_ind,
     SQLUDF_NULLIND *st3_ind,
  /* trailing arguments */
     SQLUDF_TRAIL_ARGS
  )
```

프로그램과의 오브젝트 교환: 클라이언트 변환: 이 절에서는 *클라이언트* 변환을 설명합니다. 클라이언트 변환은 구조화 유형을 클라이언트 응용프로그램과 교환합니다.

예를 들어, 다음 SQL문을 실행한다고 가정해 보십시오.

```
...
SQL TYPE IS Address_t AS VARCHAR(150) addhv;
...

EXEC SQL SELECT Address
   FROM Person
   INTO :addhv
   WHERE AGE > 25
END EXEC;
```

그림17은 그 주소를 클라이언트 프로그램으로 바인드아웃하는 프로세스를 보여줍니다.

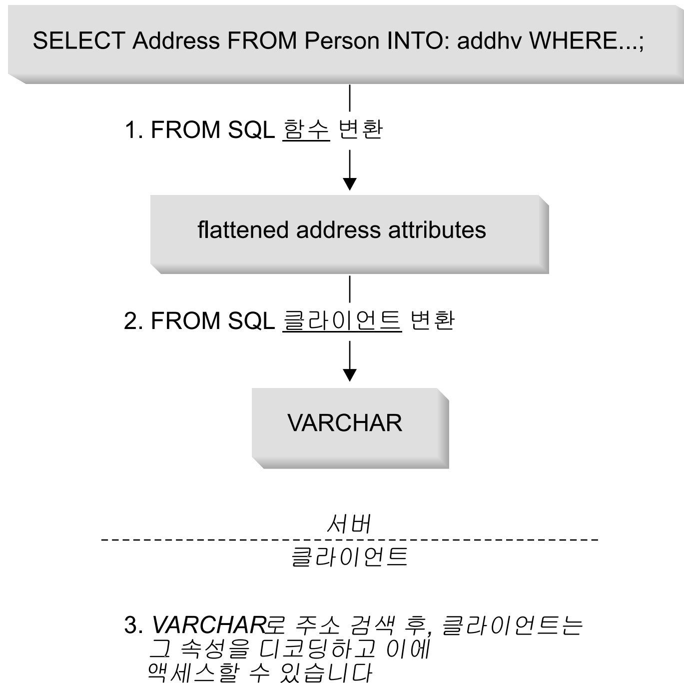


그림 17. 클라이언트 응용프로그램으로 구조화 유형 바인드아웃

1. 오브젝트를 먼저 FROM SQL 함수 변환으로 전달하여 이를 그 기본 유형 속성으로 분해해야 합니다.
2. FROM SQL 클라이언트 변환은 값을 VARCHAR 또는 BLOB과 같은 단일 내장 유형으로 인코드해야 합니다. 이를 통해 클라이언트 프로그램은 전체 값을 단일 호스트 변수로 수신합니다.

   이 인코딩은 속성을 연속 저장영역에 복사(필수 정렬에 대해 제공)하는 것처럼 간단할 수 있습니다. 속성의 인코딩과 디코딩은 일반적으로 SQL로 이루어지지 않으므로 클라이언트 변환은 보통 외부 UDF로 작성됩니다.

   플랫폼간의 데이터 처리에 대해 370 페이지의 『데이터 변환 고려사항』을 참조하십시오.
3. 클라이언트 프로그램은 값을 처리합니다.

그림18은 주소를 다시 데이터베이스로 전달하는 역 프로세스를 보여줍니다.

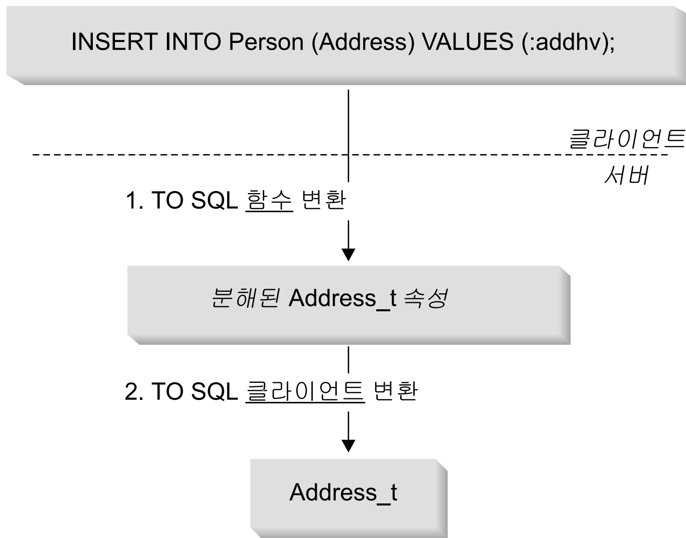


그림 18. 클라이언트에서 구조화 유형으로 바인딩

1. 클라이언트 응용프로그램은 주소를 TO SQL 클라이언트 변환에서 예상한 형식으로 인코드합니다.
2. TO SQL 클라이언트 변환은 단일 내장 유형을 TO SQL 함수 변환에 대한 입력으로 사용되는 기본 유형 속성 세트로 분해됩니다.
3. TO SQL 함수는 주소를 구성하고 이를 데이터베이스로 리턴합니다.

외부 UDF를 사용한 클라이언트 변환 구현: 클라이언트 변환을 다른 외부 UDF와 같은 방법을 등록하십시오. 예를 들어, 주소에 적합한 인코딩과 디코딩을 수행하는 외부 UDF를 작성했다고 하십시오. FROM SQL 클라이언트 변환을 from_sql_to_client로 그리고 TO SQL 클라이언트 변환을 to_sql_from_client로 명명했다고 하십시오. 이런 경우 함수의 출력은 적합한 FROM SQL과 RO SQL 함수 변환에서 입력으로 사용될 수 있는 형식으로 되어 있습니다.

```
CREATE FUNCTION from_sql_to_client (Address_t)
   RETURNS VARCHAR (150)
   LANGUAGE C
   TRANSFORM GROUP func_group
   EXTERNAL NAME 'addressudf!address_from_sql_to_client'
   NOT VARIANT
   NO EXTERNAL ACTION
   NOT FENCED
   NO SQL
   PARAMETER STYLE DB2SQL;
```

이전 예에 있는 DDL은 마치 from_sql_to_client UDF가 Address_t 유형의 매개변수를 승인하는 것처럼 보이게 만듭니다. 실제로 일어난 것은 from_sql_to_client UDF가 호출된 각 행에 대해 Addresstofunc 변환은 Address를 다양한 속성으로 분해하는 작업입니다. from_sql_to_client UDF는 간단한 문자열을 생산하고 주소 속성을 표시할 수 있도록 형식화하여 다음 단순 SQL 조회를 통해 Person 테이블의 각 행에 대해 Name과 Address 속성을 표시합니다.

```
SELECT Name, from_sql_to_client (Address)
FROM Person;
```

클라이언트에서 바인드인할 클라이언트 변환: 다음 DDL은 VARCHAR-인코드된 오브젝트를 클라이언트에서 취하고 이를 다양한 기본 유형 속성으로 분해하며 TO SQL 함수 변환으로 전달하는 함수를 등록합니다.

```
CREATE FUNCTION to_sql_from_client (VARCHAR (150))
RETURNS Address_t
   LANGUAGE C
   TRANSFORM GROUP func_group
   EXTERNAL NAME 'addressudf!address_to_sql_from_client'
   NOT VARIANT
   NO EXTERNAL ACTION
   NOT FENCED
   NO SQL
   PARAMETER STYLE DB2SQL;
```

비록 to_sql_from_client가 주소를 직접 리턴하는 것처럼 보이지만, 실제로는 to_sql_from_client가 VARCHAR (150)을 기본 유형 속성 세트로 변환하는 것입니다. 그런 후 DB2는 functoaddress TO SQL 변환을 암시적으로 호출하여 데이터베이스로 리턴되는 주소 오브젝트를 구성합니다.

DB2는 어떤 함수 변환을 호출해야 하는지 압니까? to_sql_from_client와 from_sql_to_client 양쪽에 있는 DDL에는 TRANSFORM GROUP이라는 절이 있습니다. 이 절은 DB2에게 이런 함수에서 주소 유형을 처리할 때 사용하는 변환 세트를 알려줍니다. 자세한 내용은 355 페이지의 『유형과 변환 연관』의 내용을 참조하십시오.

데이터 변환 고려사항: 서버와 클라이언트간에 데이터, 특히 2진 데이터가 교환될 때 여러 가지 데이터 변환 문제를 고려해야 합니다. 예를 들어, 데이터가 여러 바이트-정렬 스킴으로 플랫폼간에 전송되면 수치 데이터는 바이트 리버설 프로세스를 거쳐 올바른 수치 값을 복원해야 합니다. 여러 운영 체제에는 메모리에서 수치 데이터를 참조하는 특정 정렬 요구사항이 있습니다. 일부 운영 체제는 이런 요구사항이 충족되지 않을 때 프로그램 예외를 일으킵니다. 문자 데이터 유형은 데이터베이스에 의해 자동으로 변환됩니다. 단, 문자 데이터가 BLOB 또는 VARCHAR FOR BIT DATA와 같은 2진 데이터 유형에 포함될 때는 예외입니다.

데이터 변환 문제를 피할 수 있는 두 가지 방법이 있습니다.

- 항상 오브젝트를 수치 데이터를 포함한 인쇄가능한 문자 데이터 유형으로 변환합니다.

  이 접근 방법은 잠재적 변환이 많이 필요한다는 면에서의 성능 저하와 클라이언트 또는 변환 함수 자체에서 이런 오브젝트를 액세스하는 코드가 더 복잡해졌다는 단점이 있습니다.

- 라이브러리 데이터 유형으로 변환된 오브젝트에 대해 플랫폼에 무관한 형식을 개발하십시오. 이는 Java 구현에서 수행한 접근방법과 비슷합니다. 반드시 다음을 수행하십시오.
  - 이런 압축 오브젝트를 패킹하거나 패킹 해제하여 개별 데이터 유형을 제대로 인코드하거나 디코드할 때 그리고 데이터 손상이나 프로그램 결함을 피할 때 주의하십시오.
  - 변환된 유형에 헤더 정보를 충분히 포함시켜서 인코드된 오브젝트의 나머지가 클라이언트 또는 서버 플랫폼에 관계없이 제대로 해석되도록 합니다.
  - CREATE FUNCTION의 DBINFO 옵션을 사용하여 데이터베이스 서버 환경에 관련된 다양한 특성을 변환 함수에 전달합니다. 이런 특성은 플랫폼과 무관한 형식으로 헤더에 포함될 수 있습니다. DBINFO 사용에 대한 자세한 내용은 435 페이지의 『인수를 DB2에서 UDF로 전달』을 참조하십시오.

  데이터 변환에 대한 자세한 내용은 560 페이지의 『자국어 지원 고려사항』을 참조하십시오.

주: 가능한 한 서버와 클라이언트간에 일어나는 데이터 전송에 연관된 모든 복잡성을 제대로 처리할 수 있도록 변환 함수를 작성해야 합니다. 응용프로그램을 설계할 때 환경의 특정 요구사항을 고려하고 완전한 일반성과 단순성 간의 장단점을 평가하십시오. 예를 들어, 데이터베이스 서버와 그 모든 클라이언트가 AIX 환경에서 실행하고 같은 코드 페이지를 사용한다는 것을 알면 현재 변환이 필요하지 않으므로 앞에서 설명된 고려사항을 무시해야 합니다. 그러나, 환경이 미래에 변하면 데이터 변환을 올바로 처리할 수 있도록 원래 설계를 수정하는 데 상당한 노력을 기울여야 할 것입니다.

변환 함수 요약

표15는 외부 루틴 또는 클라이언트 응용프로그램으로 바인드아웃하는 여부에 따라 필요한 변환 함수를 결정할 때 도움을 주기 위한 것입니다.

표 15. 변환 함수의 특성

특성	외부 루틴과의 값 교환		클라이언트 응용프로그램과의 값 교환	
변환 방향	FROM SQL	TO SQL	FROM SQL	TO SQL
어떤 것이 변환됩니까	루틴 매개변수	루틴 결과	출력 호스트 변수	입력 호스트 변수

표 15. 변환 함수의 특성 (계속)

특성	외부 루틴과의 값 교환		클라이언트 응용프로그램과의 값 교환	
동작	분해	구성	인코드	디코드
변환 함수 매개변수	구조화 유형	내장 유형의 행	구조화 유형	하나의 내장 유형
변환 함수 결과	내장 유형의 행(속성일 것임)	구조화 유형	하나의 내장 유형	구조화 유형
다른 변환에 종속되어 있습니까?	아니오	아니오	FROM SQL UDF 변환	TO SQL UDF 변환
언제 변환 그룹이 지정됩니까?	UDF가 등록될 때		정적: 사전 처리 컴파일 시간 동적: 특수 레지스터	
데이터 변환 고려사항이 있습니까?	아니오		예	

주: 비록 일반적이지는 않지만 다음의 경우가 참일 때 클라이언트 유형변환은 실제로 SQL로 작성됩니다.

- 구조화 유형에 속성이 하나뿐입니다.
- 내장 유형으로 속성을 인코드하고 디코드하는 것은 SQL 연산자나 함수의 조합으로 이루어집니다.

이런 경우 함수 변환에 의존하여 구조화 유형의 값을 클라이언트 응용프로그램과 교환하지 않아도 됩니다.

DB2에서 부속 유형 데이터 검색(바인드아웃)

앞 절에 있는 대부분의 정보는 응용프로그램이 알려진 정확한 유형을 전달한다고 간주합니다. 데이터 모델이 부속 유형을 사용하면 컬럼에 있는 값이 여러 부속 유형 중 하나일 것입니다. 이 절에서는 실제 입력 유형을 기반으로 올바른 변환 함수를 동적으로 선택할 수 있는 방법을 설명합니다.

다음 SELECT문을 발행한다고 하십시오.

```
SELECT Address
   FROM Person
   INTO :hvaddr;
```

응용프로그램에는 `Address_t`의 인스턴스, `US_addr_t` 등이 리턴되는지 알 수 있는 방법이 없습니다. 예를 너무 복잡하지 않게 유지하기 위해 `Address_t`나

US_addr_t만 리턴될 수 있다고 가정합니다. 이런 유형의 구조는 다르므로 속성을 분해하는 변환은 달라야 합니다. 적합한 변환이 호출되도록 다음 단계를 수행하십시오.

단계 1. 각각의 주소 변형에 대해 FROM SQL 함수 변환을 작성하십시오.

```
CREATE FUNCTION addresstofunc(A address_t)
   RETURNS ROW
   (Street VARCHAR(30), Number CHAR(15), City
   VARCHAR(30), STATE VARCHAR (10))
LANGUAGE SQL
   RETURN VALUES
   (A..Street, A..Number, A..City, A..State)

CREATE FUNCTION US_addresstofunc(A US_addr_t)
   RETURNS ROW
   (Street VARCHAR(30), Number CHAR(15), City
   VARCHAR(30), STATE VARCHAR (10), Zip
   CHAR(10))
LANGUAGE SQL
   RETURN VALUES
   (A..Street, A..Number, A..City, A..State, A..Zip)
```

단계 2. 각 유형 변형마다 변환 그룹을 하나씩 작성하십시오.

```
CREATE TRANSFORM FOR Address_t
   funcgroup1 (FROM SQL WITH FUNCTION addresstofunc)

CREATE TRANSFORM FOR US_addr_t
   funcgroup2 (FROM SQL WITH FUNCTION US_addresstofunc)
```

단계 3. 각 유형 변형마다 외부 UDF를 작성하십시오.

Address_t 유형에 대해 외부 UDF를 등록하십시오.

```
CREATE FUNCTION address_to_client (A Address_t)
   RETURNS VARCHAR(150)
   LANGUAGE C
   EXTERNAL NAME 'addressudf!address_to_client'
   ...
   TRANSFORM GROUP funcgroup1
```

address_to_client UDF를 작성하십시오.

```
void SQL_API_FN address_to_client(
   SQLUDF_VARCHAR  *street,
   SQLUDF_CHAR     *number,
   SQLUDF_VARCHAR  *city,
   SQLUDF_VARCHAR  *state,
```

```
      SQLUDF_VARCHAR  *output,

      /* Null indicators for attributes */
      SQLUDF_NULLIND *street_ind,
      SQLUDF_NULLIND  *number_ind,
      SQLUDF_NULLIND  *city_ind,
      SQLUDF_NULLIND  *state_ind,
      /* Null indicator for instance */
      SQLUDF_NULLIND *address_ind,
      /* Null indicator for output */
      SQLUDF_NULLIND  *output_ind,
      SQLUDF_TRAIL_ARGS )

   {
      sprintf (output, "[address_t] [Street:%s] [number:%s]
      [city:%s] [state:%s]",
      street, number, city, state);
      *output_ind = 0;
   }
```

US_addr_t 유형에 대해 외부 UDF를 등록하십시오.

```
CREATE FUNCTION address_to_client (A US_addr_t)
   RETURNS VARCHAR(150)
   LANGUAGE C
   EXTERNAL NAME 'addressudf!US_addr_to_client'
   ...
   TRANSFORM GROUP funcgroup2
```

US_addr_to_client UDF를 작성하십시오.

```
void SQL_API_FN US_address_to_client(
   SQLUDF_VARCHAR  *street,
   SQLUDF_CHAR     *number,
   SQLUDF_VARCHAR  *city,
   SQLUDF_VARCHAR  *state,
   SQLUDF_CHAR     *zip,
   SQLUDF_VARCHAR  *output,

   /* Null indicators */
   SQLUDF_NULLIND *street_ind,
   SQLUDF_NULLIND  *number_ind,
   SQLUDF_NULLIND  *city_ind,
   SQLUDF_NULLIND  *state_ind,
   SQLUDF_NULLIND *zip_ind,
   SQLUDF_NULLIND  *us_address_ind,
   SQLUDF_NULLIND  *output_ind,
   SQLUDF_TRAIL_ARGS )
```

```
{
   sprintf (output, "[US_addr_t] [Street:%s] [number:%s]
   [city:%s] [state:%s] [zip:%s]",
   street, number, city, state, zip);
   *output_ind = 0;
}
```

단계 4. 인스턴스를 처리하기 위해 올바른 외부 UDF를 선택하는 SQL 본문 UDF를 작성하십시오. 다음 UDF는 UNION ALL절에 의해 조합된 SELECT문 내의 TREAT 스펙을 사용하여 올바른 FROM SQL 변환을 호출하고 결과를 임시 테이블에 넣습니다.

```
CREATE FUNCTION addr_stream (ab Address_t)
   RETURNS VARCHAR(150)
   LANGUAGE SQL
   RETURN
   WITH temp(addr) AS
   (SELECT address_to_client(ta.a)
      FROM TABLE (VALUES (ab)) AS ta(a)
      WHERE ta.a IS OF (ONLY Address_t)
      UNION ALL
   SELECT address_to_client(TREAT (tb.a AS US_addr_t))
      FROM TABLE (VALUES (ab)) AS tb(a)
      WHERE tb.a IS OF (ONLY US_addr_t))
   SELECT addr FROM temp;
```

이 때 응용프로그램은 Addr_stream 함수를 호출하여 적합한 외부 UDF를 호출할 수 있습니다.

```
SELECT Addr_stream(Address)
   FROM Employee;
```

단계 5. Addr_stream 외부 UDF를 Address_t의 FROM SQL *클라이언트* 변환으로 추가하십시오.

```
CREATE TRANSFORM GROUP FOR Address_t
   client_group (FROM SQL
   WITH FUNCTION Addr_stream)
```

주: 응용프로그램에서 type 술어를 사용하여 조회에서 특정 주소 유형을 지정하면 US_addr_t에 대해 Addr_stream을 FROM SQL로 클라이언트 변환에 추가하십시오. 이렇게 하면 조회가 특별히 US_addr_t의 인스턴스를 요청할 때 Addr_stream이 호출됩니다.

단계 6. 응용프로그램을 TRANSFORM GROUP 옵션 세트와 함께 client_group 으로 바인드하십시오.

```
PREP myprogram TRANSFORM GROUP client_group
```

DB2가 SELECT Address FROM Person INTO :hvar문이 들어 있는 응용프로그램을 바인드하면 DB2는 FROM SQL 클라이언트 변환을 찾습니다. DB2는 구조화 유형이 바인드아웃되었음을 인식하고 client_group 변환 그룹을 살펴봅니다. 왜냐하면 이는 6의 바인드 시간에 지정되는 변환 그룹이기 때문입니다.

변환 그룹에는 375 페이지의 5의 Address_t 루트 유형에 연관된 Addr_stream 변환 함수가 들어 있습니다. Addr_stream은 375 페이지의 4에 정의된 SQL 본문 함수이므로 다른 변환 함수에 대한 종속성이 없습니다. Addr_stream 함수는 :hvaddr 호스트 변수에서 필요한 데이터 유형인 VARCHAR(150)을 리턴합니다.

Addr_stream 함수는 이 예에서 US_addr_t로 대체될 수 있는 Address_t 유형의 입력 값을 취하여 입력 값의 동적 유형을 결정합니다. Addr_stream이동적 유형을 결정하면 이는 값에 대해 해당 외부 UDF를 호출합니다. 동적 유형이 Address_t이면 address_to_client 또는 동적 유형이 US_addr_t이면 USaddr_to_client입니다. 이런 두 UDF는 373 페이지의 3에서 정의됩니다. 각 UDF는 각각의 구조화 유형을 Addr_stream 변환 함수에서 필요한 유형인 VARCHAR(150)로 분해합니다.

구조화 유형을 입력으로 승인하려면 각 UDF는 FROM SQL 변환 함수로 입력 구조화 유형 인스턴스를 개별 속성 매개변수로 분해해야 합니다. 373 페이지의 3의 CREAYE FUNCTION문은 이런 변환이 들어 있는 TRANSFORM GROUP을 명명합니다.

변환 함수의 CREATE FUNCTION문은 373 페이지의 1에서 발행됩니다. 변환 함수를 그 변환 그룹에 연관시키는 CREATE FUNCTION문은 373 페이지의 2에서 발행됩니다.

DB2로 부속 유형 데이터 리턴(바인드인)

우선 372 페이지의 『DB2에서 부속 유형 데이터 검색(바인드아웃)』에서 설명된 응용프로그램이 주소 값을 처리하면, 이는 변경된 값을 다시 데이터베이스에 삽입해야 합니다. 다음 구문을 사용하여 응용프로그램에서 구조화 유형을 DB2 데이터베이스에 삽입한다고 가정해 보십시오.

```
INSERT INTO person (Oid, Name, Address)
   VALUES ('n', 'Norm', :hvaddr);
```

구조화 유형에 대해 INSERT문을 실행하려면 응용프로그램은 다음 단계를 수행해야 합니다.

단계 1. 주소의 각 변형에 대해 TO SQL 함수 변환을 작성하십시오. 다음 예는 Address_t와 US_addr_t 유형을 변환하는 SQL-본문 UDF를 보여줍니다.

```
CREATE FUNCTION functoaddress
 (str VARCHAR(30), num CHAR(15), cy VARCHAR(30), st VARCHAR (10))
RETURNS Address_t
LANGUAGE SQL
RETURN Address_t()..street(str)..number(num)..city(cy)..state(st);

CREATE FUNCTION functoaddress
 (str VARCHAR(30), num CHAR(15), cy VARCHAR(30), st VARCHAR (10),
  zp CHAR(10))
RETURNS US_addr_t
LANGUAGE SQL
RETURN US_addr_t()..street(str)..number(num)..city(cy)
    ..state(st)..zip(zp);
```

단계 2. 각 유형 변형마다 변환 그룹을 하나씩 작성하십시오.

```
CREATE TRANSFORM FOR Address_t
   funcgroup1 (TO SQL
   WITH FUNCTION functoaddress);

CREATE TRANSFORM FOR US_addr_t
   funcgroup2 (TO SQL
   WITH FUNCTION functousaddr);
```

단계 3. 인코드된 주소 유형을 각 유형 변형마다 하나씩 리턴하는 외부 UDF를 작성하십시오.

Address_t 유형에 대해 외부 UDF를 등록하십시오.

```
CREATE FUNCTION client_to_address (encoding VARCHAR(150))
RETURNS Address_t
   LANGUAGE C
```

```
TRANSFORM GROUP funcgroup1
...
EXTERNAL NAME 'address!client_to_address';
```

client_to_address의 Address_t 버전에 대해 외부 UDF를 작성하십시오.

```
void SQL_API_FN client_to_address (
   SQLUDF_VARCHAR *encoding,
   SQLUDF_VARCHAR  *street,
   SQLUDF_CHAR     *number,
   SQLUDF_VARCHAR  *city,
   SQLUDF_VARCHAR  *state,

   /* Null indicators */
   SQLUDF_NULLIND *encoding_ind,
   SQLUDF_NULLIND *street_ind,
   SQLUDF_NULLIND  *number_ind,
   SQLUDF_NULLIND  *city_ind,
   SQLUDF_NULLIND  *state_ind,
   SQLUDF_NULLIND *address_ind,
   SQLUDF_TRAIL_ARGS )
{
   char c[150];
   char *pc;

   strcpy(c, encoding);

   pc = strtok (c, ":]");
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (street, pc);
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (number, pc);
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (city, pc);
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (state, pc);

   *street_ind = *number_ind = *city_ind
   = *state_ind = *address_ind = 0;
}
```

US_addr_t 유형에 대해 외부 UDF를 등록하십시오.:

```
CREATE FUNCTION client_to_us_address (encoding VARCHAR(150))
RETURNS US_addr_t
   LANGUAGE C
   TRANSFORM GROUP funcgroup1
   ...
   EXTERNAL NAME 'address!client_to_US_addr';
```

client_to_address의 US_addr_t 버전에 대해 외부 UDF를 작성하십시오.

```
void SQL_API_FN client_to_US_addr(
   SQLUDF_VARCHAR *encoding,
   SQLUDF_VARCHAR  *street,
   SQLUDF_CHAR     *number,
   SQLUDF_VARCHAR  *city,
   SQLUDF_VARCHAR  *state,
   SQLUDF_VARCHAR *zip,

   /* Null indicators */
   SQLUDF_NULLIND *encoding_ind,
   SQLUDF_NULLIND *street_ind,
   SQLUDF_NULLIND  *number_ind,
   SQLUDF_NULLIND  *city_ind,
   SQLUDF_NULLIND  *state_ind,
   SQLUDF_NULLIND *zip_ind,
   SQLUDF_NULLIND *us_addr_ind,
   SQLUDF_TRAIL_ARGS )

{
   char c[150];
   char *pc;

   strcpy(c, encoding);

   pc = strtok (c, ":]");
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (street, pc);
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strncpy (number, pc,14);
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (city, pc);
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (state, pc);
```

```
        pc = strtok (NULL, ":]");
        pc = strtok (NULL, ":]");
        strncpy (zip, pc, 9);

        *street_ind = *number_ind = *city_ind
        = *state_ind = *zip_ind = *us_addr_ind = 0;
     }
```

단계 4. 인스턴스를 처리하기 위해 올바른 외부 UDF를 선택하는 SQL-본문 UDF를 작성하십시오. 다음 UDF는 TYPE 술어를 사용하여 올바른 클라이언트 변환을 호출합니다. 결과는 임시 테이블에 보관됩니다.

```
CREATE FUNCTION stream_address (ENCODING VARCHAR(150))
RETURNS Address_t
LANGUAGE SQL
   RETURN
   (CASE(SUBSTR(ENCODING,2,POSSTR(ENCODING,']')-2))
   WHEN 'address_t'
      THEN client_to_address(ENCODING)
   WHEN 'us_addr_t'
      THEN client_to_us_addr(ENCODING)
   ELSE NULL
   END);
```

단계 5. Address_t에 대해 stream_address UDF를 TO SQL 클라이언트 변환으로 추가하십시오.

```
CREATE TRANSFORM FOR Address_t
   client_group (TO SQL
   WITH FUNCTION stream_address);
```

단계 6. 응용프로그램을 TRANSFORM GROUP 옵션 세트와 함께 client_group으로 바인드하십시오.

```
PREP myProgram2 TRANSFORM GROUP client_group
```

구조화 유형의 INSERT문이 있는 응용프로그램이 바인드되면 DB2는 TO SQL 클라이언트 변환을 찾습니다. DB2는 client_group 변환 그룹에서 변환을 찾습니다. 왜냐하면 이는 6의 바인드 시간에 지정되는 변환 그룹이기 때문입니다. DB2는 필요한 변환 함수를 찾습니다. 이는 5의 Address_t 루트 유형에 연관된 stream_address입니다.

stream_address는 4에서 정의된 SQL 본문 함수이므로 이는 추가 변환 함수에 대한 종속성이 없습니다. 입력 매개변수에 대해 stream_address는 :hvaddr 응

용프로그램 호스트 변수에 해당하는 VARCHAR(150)을 승인합니다. stream_address는 올바른 루트 유형인 Address_t와 올바른 동적 유형으로 값을 리턴합니다.

stream_address는 동적 유형을 명명하는 부속 문자열에 대해 VARCHAR(150) 입력 매개변수를 분석합니다. 이 경우 'Address_t' 또는 'US_addr_t' 입니다. 그러면 stream_address는 해당 외부 UDF를 호출하여 VARCHAR(150)을 분석하고 지정된 유형의 오브젝트를 리턴합니다. 두 가지 client_to_address() UDF와 가능한 각 유형을 리턴하기 위한 것 하나가 있습니다. 이런 UDF는 377 페이지의 3에서 정의됩니다. 각 UDF는 입력 VARCHAR(150)을 취하고 내부적으로 적합한 구조화 유형의 속성을 구성하여 구조화 유형을 리턴합니다.

구조화 유형을 리턴하려면 각 UDF는 출력 속성 값을 구조화 유형의 인스턴스로 구성하는 TO SQL 변환 함수가 필요합니다. 377 페이지의 3의 CREATE FUNCTION문은 변환이 들어 있는 TRANSFORM GROUP을 명명합니다.

377 페이지의 1의 SQL 본문 함수와 377 페이지의 2의 변환 그룹과의 연관은 377 페이지의 3의 CREATE FUNCTION문에서 명명됩니다.

구조화 유형 호스트 변수 작업

구조화 유형 호스트 변수 선언

정적 SQL에서 구조화 유형 호스트 변수를 수신하거나 전송하려면 구조화 유형을 표현할 때 사용된 내장 유형을 나타내는 SQL 선언을 제공해야 합니다. 선언의 형식은 다음과 같습니다.

```
EXEC SQL BEGIN DECLARE SECTION ;

  SQL TYPE IS structured_type AS base_type host-variable-name ;

EXEC SQL END DECLARE SECTION;
```

예를 들어, Address_t유형이 클라이언트 응용프로그램으로 전달될 때 가변-길이의 문자 유형으로 변환되어야 한다고 가정하십시오. Address_t 유형 호스트 변수에 대한 다음 선언을 사용하십시오.

```
SQL TYPE IS Address_t AS VARCHAR(150) addrhv;
```

구조화 유형 설명

구조화 유형 변수가 있는 명령문의 DESCRIBE는 DB2에서 FROM SQL 변환 함수의 결과 유형 설명을 SQLDA의 기본 SQLVAR의 SQLTYPE 필드에 놓게 합니다. 그러나, CURRENT DEFAULT TRANSFORM GROUP 특수 레지스터를 사용하여 지정된 TRANSFORM GROUP이 없거나 명명된 그룹에 정의된 FROM SQL 변환 함수가 없으므로 정의된 FROM SQL 변환 함수가 없으면 DESCRIBE는 오류를 리턴합니다.

구조화 유형의 실제 이름은 SQLVAR2에 리턴됩니다. SQLDA의 구조에 대한 자세한 내용은 *SQL 참조서*를 참조하십시오.

제13장 대형 오브젝트(LOB) 사용

LOB란? 383
대형 오브젝트(LOB) 데이터 유형(BLOB, CLOB, DBCLOB)의 이해 384
대형 오브젝트(LOB) 위치 지정자의 이해 386
예: CLOB 값에 대해 작업할 위치 지정자 사용 388
샘플 LOBLOC 프로그램 작업 방법 . . 388
C 예: LOBLOC.SQC 389
COBOL 예: LOBLOC.SQB 391
예: LOB 표현식 평가 연기 394
샘플 LOBEVAL 프로그램 작업 방법 395
C 예: LOBEVAL.SQC 396
COBOL 예: LOBEVAL.SQB 398
표시기 변수 및 LOB 위치 지정자 . . . 401
LOB 파일 참조 변수 401
예: 파일로 문서 발췌 403
샘플 LOBFILE 프로그램 작업 방법 . . 403
C 예: LOBFILE.SQC 404
COBOL 예: LOBFILE.SQB 406
예: CLOB 컬럼에 데이터 삽입 408

LOB란?

LONG VARCHAR 및 LONG VARGRAPHIC 데이터 유형의 저장영역 한계는 32K 바이트입니다. 이 유형은 소형 또는 중형 텍스트 데이터용으로 충분하나 응용프로그램은 대형 텍스트 문서를 저장해야 할 경우가 있습니다. 오디오, 비디오, 드로잉, 텍스트 및 그래픽의 혼합, 이미지 등의 다양한 추가 데이터 유형을 저장해야 할 수도 있습니다. DB2는 최고 2GB 크기의 문자열로서 이러한 데이터 오브젝트를 저장하기 위해 세 가지 데이터 유형을 제공합니다. 세 가지 데이터 유형은, 2진 대형 오브젝트(BLOB), 1바이트 문자 대형 오브젝트(BLOB) 및 2바이트 문자 대형 오브젝트(CLOB)를 말합니다.

대형 오브젝트(LOB) 저장과 함께 데이터베이스에서 각 LOB를 참조, 사용 및 수정하는 방법도 필요합니다. 각 DB2 테이블에는 많은 양의 관련 LOB 데이터가 있을 수 있습니다. 단일 LOB 값이 2GB를 초과할 수 없지만 한 행에는 LOB 데이터가 최고 24GB 포함될 수 있으며 한 테이블에는 LOB 데이터가 최고 4TB가 포함될 수 있습니다. 어떤 시점에 특정 행의 LOB 컬럼 내용에 *대형 오브젝트(LOB) 값*이 있습니다.

다른 데이터 유형에서와 마찬가지로 호스트 변수를 사용하여 LOB를 참조 및 조작할 수 있습니다. 그러나, 호스트 변수는 LOB 값을 보유할 만큼 충분치 못한 클라이언트 메모리 버퍼를 사용합니다. 이러한 대형 값을 조작하는 데 있어서 다른

© Copyright IBM Corp. 1993, 2000

방법들이 필수적입니다. *위치 지정자*는 LOB 값 조각들을 추출할 때와 데이터베이스 서버의 대형 오브젝트(LOB) 값을 식별 및 조작할 때 유용합니다. *파일 참조 변수*는 대형 오브젝트(LOB) 값(또는 대형 부분)을 클라이언트로부터 또는 클라이언트로 물리적으로 이동시키는 데 유용합니다.

주: DB2는 JDBC 및 SQLJ 응용프로그램에 대한 LOB 지원을 제공합니다. Java 응용프로그램에서의 LOB 사용에 대해서는 723 페이지의 『JDBC 2.0』의 내용을 참조하십시오.

다음 절에서는 위에서 소개한 주제들에 대해 보다 상세하게 논의합니다.

대형 오브젝트(LOB) 데이터 유형(BLOB, CLOB, DBCLOB)의 이해

대형 오브젝트(LOB) 데이터 유형은 *크기*가 0바이트에서 2GB-1인 데이터를 저장합니다.

세 가지 대형 오브젝트(LOB) 데이터 유형의 정의는 다음과 같습니다.

- 문자 대형 오브젝트(CLOB) — 연관된 코드 페이지가 있는 1바이트 문자로 구성된 문자열. 정보량이 일반 VARCHAR 데이터 유형의 한계(상한값은 4K 바이트)를 넘어설 수 있는 텍스트 지향 정보를 보유하는 데 가장 적합합니다. 다른 문자 유형과의 호환성 뿐만 아니라 정보의 코드 페이지 변환도 지원됩니다.
- 2바이트 문자 대형 오브젝트(DBCLOB) — 연관된 코드 페이지가 있는 2바이트 문자들로 구성된 문자열. 이 데이터 유형은 2바이트 문자 세트가 사용되는 텍스트 지향 정보를 보유하는 데 가장 적합합니다. 다른 문자 유형과의 호환성 뿐만 아니라 정보의 코드 페이지 변환이 여기에서도 지원됩니다.
- 2진 대형 오브젝트(BLOB) — 연관된 코드 페이지가 없는 바이트들로 구성된 2진 문자열. 이 데이터 유형은 2진 데이터를 저장할 수 있으므로, 사용자 정의 구별 유형(UDT)이 사용하기에 완벽한 소스 유형이 되어 가장 유용할 수 있습니다. 소스 유형으로서 BLOB를 사용하는 UDT는 이미지, 음성, 그래픽 및 기타 유형의 비지니스 또는 응용프로그램별 데이터를 저장하기 위해 작성됩니다. UDT에 대한 세부사항은 303 페이지의 『제11장 사용자 정의 구별 유형』의 내용을 참조하십시오.

분리된 데이터베이스 위치는 모든 대형 오브젝트(LOB) 값을 테이블의 레코드 외부에 저장합니다. 테이블의 각 행에 있는 각 대형 오브젝트(LOB)에 대해 하나의 *대형 오브젝트(LOB) 설명자*가 있습니다. 대형 오브젝트(LOB) 설명자에는 디스크의 다른 위치에 저장된 대형 오브젝트(LOB) 데이터에 액세스하는 데 사용되는 제어 정보가 들어 있습니다. 이는 LOB *크기*가 2GB가 되도록 하는 레코드 외부의 대형 오브젝트(LOB) 데이터를 저장하는 것입니다. 대형 오브젝트(LOB) 설명자에 액세스하면 LOB를 조작할 때 소량의 오버헤드가 발생합니다.(저장영역 및 성능상의 이유로 소형 데이터 항목을 LOB에 두지 않으려 할 수 있습니다.)

각 대형 오브젝트(LOB) 컬럼의 최대 *크기*는 CREATE TABLE문에 있는 대형 오브젝트(LOB) 유형의 선언 일부입니다. 대형 오브젝트(LOB) 컬럼의 최대 *크기*는 해당 컬럼에 있는 LOB 설명자의 최대 *크기*를 결정합니다. 결과적으로 모든 데이터 유형의 많은 컬럼들이 단일 행에 어떻게 들어갈 수 있는지를 결정합니다. 행의 LOB 설명자에 의해 사용되는 공간의 범위는 해당 컬럼의 최대 *크기*에 따라 약 60바이트에서 300바이트까지입니다. LOB 설명자의 특정 *크기*에 대해서는 *SQL 참조서*에 있는 CREATE TABLE문을 참조하십시오.

CREATE TABLE의 lob-options-clause는 LOB 컬럼의 변경사항의 로그 여부를 사용자가 선택할 수 있도록 합니다. 또한, 이 절은 LOB 설명자에 대해 간결한 표시를 허용하기도 합니다. 이는 LOB를 저장할 충분한 공간만 할당할 수 있거나, LOB에 대한 이후의 첨부 조작을 위한 추가 공간을 할당할 수 있음을 의미합니다. tablespace-options-clause는 사용자가 long 필드 또는 LOB 데이터 유형의 컬럼 값을 저장할 LONG 테이블 공간을 식별할 수 있도록 합니다. CREATE TABLE문 및 ALTER TABLE문에 대한 세부사항은 *SQL 참조서*의 내용을 참조하십시오.

잠재적으로 초대형의 *크기*를 가질 수 있는 LOB는 데이터베이스 외부로 이동될 때 데이터베이스 시스템 성능을 현저히 떨어뜨릴 수 있습니다. DB2가 1GB보다 큰 LOB 값의 로그를 허용하지 않아도, 수백 메가바이트의 LOB 값은 데이터베이스 로그를 거의 최대 용량으로 신속하게 채울 수 있습니다. SQLCODE -355(SQLSTATE 42993) 오류는 1GB보다 큰 LOB를 로그하려 한 결과 발생합니다. CREATE TABLE문 및 ALTER TABLE문의 lob-options-clause를 통해 특정 LOB 컬럼의 로깅을 중지할 수 있습니다. 옵션을 NOT LOGGED로 설정한 결과

성능이 향상되어도, 최신 백업 이후의 LOB 값의 변경사항은 롤 포워드 복구 중에 소실됩니다. 이러한 주제에 대해서는 *관리 안내서*의 내용을 참조하십시오.

대형 오브젝트(LOB) 위치 지정자의 이해

이론적으로 LOB 위치 지정자는 잠시 동안 있다는 간단한 개념을 나타냅니다. 훨씬 큰 값을 참조하려면 쉽게 관리되는 작은 값을 사용하십시오. 구체적으로 말해서 LOB 위치 지정자는, 프로그램이 데이터베이스 시스템에 보유된 LOB 값(또는 LOB 표현식)을 참조하는 데 사용할 수 있는 호스트 변수에 저장된 4바이트 값입니다. LOB 위치 지정자를 사용하여 프로그램은 LOB 값이 일반 호스트 변수에 저장된 것처럼 LOB 값을 조작할 수 있습니다. LOB 위치 지정자 사용에 있어서의 차이점은 서버에서 응용프로그램으로(또는 다시 원래대로) LOB 값을 전송할 필요가 없습니다.

LOB 위치 지정자는 LOB 값 또는 LOB 표현식과 연결되어 있으며, 데이터베이스의 행 또는 실제 저장영역 위치가 아닙니다. 따라서 LOB 값을 위치 지정자로 선택한 후에는 위치 지정자에 의해 참조되는 값에 영향을 미친 원래의 행 또는 테이블에서 수행할 수 있는 조작이 없습니다. 위치 지정자와 연결된 값은 작업 단위(UOW)가 종료되거나 어떤 것이 처음에 오든, 위치 지정자가 명시적으로 비워질 때까지 유효합니다. FREE LOCATOR문은 연결된 값으로부터 위치 지정자를 릴리스합니다. 마찬가지 방식으로 확약 또는 구간 복원 조작은 트랜잭션과 연관된 모든 LOB 위치 지정자를 비웁니다.

LOB 위치 지정자는 DB2와 UDF 사이에도 전달될 수 있습니다. UDF가 LOB 위치 지정자를 사용하여 LOB 값을 조작 조작하는 데 사용할 수 있는 특수 API가 있습니다. 이 API에 대한 세부사항은 493 페이지의 『UDF 매개변수 또는 결과로 LOB 위치 지정자 사용』의 내용을 참조하십시오.

LOB 값을 선택할 때 다음 세 가지 옵션이 있습니다.

- 전체 LOB 값을 호스트 변수로 선택하십시오. 전체 LOB 값이 서버에서 클라이언트로 복사됩니다.
- LOB 위치 지정자만 호스트 변수로 선택하십시오. LOB 값은 그대로 서버에 남아 있고, LOB 위치 지정자는 클라이언트로 이동됩니다.

- 전체 LOB 값을 파일 참조 변수로 선택하십시오. LOB 값은 응용프로그램의 메모리를 거치지 않고 클라이언트의 파일로 이동됩니다.

프로그램 내에서 LOB 값을 사용함으로써 프로그래머가 어떤 방법이 가장 적합한지 결정하는 데 도움이 됩니다. LOB 값이 매우 크고 한 개 이상의 후속 SQL문에 대한 입력으로서만 필요한 경우 위치 지정자에 값을 보유하는 것이 가장 좋습니다. 위치 지정자를 사용하면 LOB 값을 호스트 변수로, 다시 서버로 전송하는 데 필요한 클라이언트/서버 통신량이 감소됩니다.

크기와 상관없이 프로그램이 전체 LOB 값을 필요로 할 경우 LOB를 전송하는 것 외에는 선택의 여지가 없습니다. 이런 경우에도 여전히 세 개의 옵션을 사용할 수 있습니다. 일반 또는 파일 호스트 변수로 전체 값을 선택할 수 있지만 LOB 값을 위치 지정자로 선택하고, 다음 예에서 제시된 것처럼 위치 지정자를 일반 호스트 변수로 부분별로 읽을 때 더 잘 작동됩니다.

예: CLOB 값에 대해 작업할 위치 지정자 사용

이 예에서 응용프로그램은 LOB 값에 대한 위치 지정자를 검색한 후, 위치 지정자를 사용하여 LOB 값으로부터 데이터를 추출합니다. 이 방법을 사용하여 프로그램은 LOB 데이터 한 조각에 필요한 저장영역(크기는 프로그램에 의해 결정됨)만 할당하고, 커서를 사용하여 하나의 페치 호출만 발행해야 합니다.

샘플 LOBLOC 프로그램 작업 방법

1. **호스트 변수 정의.** BEGIN DECLARE SECTION 및 END DECLARE SECTION문은 호스트 변수 선언을 분리합니다. 호스트 변수는 SQL문에서 참조될 때 콜론(:)으로 시작됩니다. CLOB LOCATOR 호스트 변수가 선언됩니다.
2. **LOB 값을 LOCATOR 호스트 변수로 페치.** CURSOR 및 FETCH 루틴을 사용하여 데이터베이스에서의 LOB 필드 위치를 호스트 변수 위치 지정자로 확보합니다.
3. **LOB LOCATORS 비우기.** 이 예에서 사용하는 LOB LOCATORS는 이전에 연결된 값으로부터 위치 지정자를 릴리스하여 비워집니다.

CHECKERR 매크로/함수는 오류 체크 유틸리티입니다. 이 오류 체크 유틸리티의 위치는 사용된 프로그래밍 언어에 따라 달라질 수 있습니다.

C
DB2 API를 호출하는 C 프로그램에 대해서는 utilapi.c에서 sqlInfoPrint 함수는 utilapi.h에서 API_SQL_CHECK로서 참조됩니다. C embedded SQL 프로그램에 대해서는 utilemb.sqc에서 sqlInfoPrint 함수는 utilemb.h에서 EMB_SQL_CHECK로서 참조됩니다.

COBOL
CHECKERR은 checkerr.cbl라고 하는 외부 프로그램입니다.

FORTRAN
CHECKERR은 util.f 파일에 있는 서브루틴입니다.

이 오류 체크 유틸리티에 대한 소스 코드에 대해서는 137 페이지의『예제 프로그램에서 GET ERROR MESSAGE 사용』의 내용을 참조하십시오.

C 예: LOBLOC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION; 1
      char number[7];
      sqlint32 deptInfoBeginLoc;
      sqlint32 deptInfoEndLoc;
      SQL TYPE IS CLOB_LOCATOR resume;
      SQL TYPE IS CLOB_LOCATOR deptBuffer;
      short lobind;
      char buffer[1000]="";
      char userid[9];
      char passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: LOBLOC\n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
          EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else {
      printf ("\nUSAGE: lobloc [userid passwd]\n\n");
      return 1;
   } /* endif */

   /* Employee A10030 is not included in the following select, because
      the lobeval program manipulates the record for A10030 so that it is
      not compatible with lobloc */

   EXEC SQL DECLARE c1 CURSOR FOR
            SELECT empno, resume FROM emp_resume WHERE resume_format='ascii'
            AND empno <> 'A00130';

   EXEC SQL OPEN c1;
   EMB_SQL_CHECK("OPEN CURSOR");

   do {
      EXEC SQL FETCH c1 INTO :number, :resume :lobind;  2
      if (SQLCODE != 0) break;
      if (lobind < 0) {
         printf ("NULL LOB indicated\n");
      } else {
         /* EVALUATE the LOB LOCATOR */
         /* Locate the beginning of "Department Information" section */
         EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
            INTO :deptInfoBeginLoc;
```

```
         EMB_SQL_CHECK("VALUES1");

         /* Locate the beginning of "Education" section (end of "Dept.Info" */
         EXEC SQL VALUES (POSSTR(:resume, 'Education'))
            INTO :deptInfoEndLoc;
         EMB_SQL_CHECK("VALUES2");

         /* Obtain ONLY the "Department Information" section by using SUBSTR */
         EXEC SQL VALUES(SUBSTR(:resume, :deptInfoBeginLoc,
            :deptInfoEndLoc - :deptInfoBeginLoc)) INTO :deptBuffer;
         EMB_SQL_CHECK("VALUES3");

         /* Append the "Department Information" section to the :buffer var. */
         EXEC SQL VALUES(:buffer || :deptBuffer) INTO :buffer;
         EMB_SQL_CHECK("VALUES4");
      } /* endif */
   } while ( 1 );

   printf ("%s\n",buffer);

   EXEC SQL FREE LOCATOR :resume, :deptBuffer;  3
   EMB_SQL_CHECK("FREE LOCATOR");

   EXEC SQL CLOSE c1;
   EMB_SQL_CHECK("CLOSE CURSOR");

   EXEC SQL CONNECT RESET;
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}
/* end of program : LOBLOC.SQC */
```

COBOL 예: LOBLOC.SQB

```
 Identification Division.
 Program-ID. "lobloc".

 Data Division.
 Working-Storage Section.

     copy "sqlenv.cbl".
     copy "sql.cbl".
     copy "sqlca.cbl".

     EXEC SQL BEGIN DECLARE SECTION END-EXEC. 1
 01 userid            pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
   49 passwd-name     pic x(18).
 01 empnum            pic x(6).
 01 di-begin-loc      pic s9(9) comp-5.
 01 di-end-loc        pic s9(9) comp-5.
 01 resume            USAGE IS SQL TYPE IS CLOB-LOCATOR.
 01 di-buffer         USAGE IS SQL TYPE IS CLOB-LOCATOR.
 01 lobind            pic s9(4) comp-5.
 01 buffer            USAGE IS SQL TYPE IS CLOB(1K).
     EXEC SQL END DECLARE SECTION END-EXEC.

 77 errloc          pic x(80).

 Procedure Division.
 Main Section.
     display "Sample COBOL program: LOBLOC".

* Get database connection information.
     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR
* format with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

* Employee A10030 is not included in the following select, because
* the lobeval program manipulates the record for A10030 so that it is
* not compatible with lobloc

     EXEC SQL DECLARE c1 CURSOR FOR
              SELECT empno, resume FROM emp_resume
```

```
              WHERE resume_format = 'ascii'
              AND empno <> 'A00130' END-EXEC.

     EXEC SQL OPEN c1 END-EXEC.
     move "OPEN CURSOR" to errloc.
     call "checkerr" using SQLCA errloc.

     Move 0 to buffer-length.

     perform Fetch-Loop thru End-Fetch-Loop
        until SQLCODE not equal 0.

* display contents of the buffer.
     display buffer-data(1:buffer-length).

     EXEC SQL FREE LOCATOR :resume, :di-buffer END-EXEC.  3
     move "FREE LOCATOR" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL CLOSE c1 END-EXEC.
     move "CLOSE CURSOR" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL CONNECT RESET END-EXEC.
     move "CONNECT RESET" to errloc.
     call "checkerr" using SQLCA errloc.
 End-Main.
        go to End-Prog.

 Fetch-Loop Section.
     EXEC SQL FETCH c1 INTO :empnum, :resume :lobind   2
        END-EXEC.

     if SQLCODE not equal 0
        go to End-Fetch-Loop.

* check to see if the host variable indicator returns NULL.
     if lobind less than 0 go to NULL-lob-indicated.

* Value exists.  Evaluate the LOB locator.
* Locate the beginning of "Department Information" section.
     EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
              INTO :di-begin-loc END-EXEC.
     move "VALUES1" to errloc.
     call "checkerr" using SQLCA errloc.

* Locate the beginning of "Education" section (end of Dept.Info)
     EXEC SQL VALUES (POSSTR(:resume, 'Education'))
               INTO :di-end-loc END-EXEC.
     move "VALUES2" to errloc.
     call "checkerr" using SQLCA errloc.

     subtract di-begin-loc from di-end-loc.

* Obtain ONLY the "Department Information" section by using SUBSTR
     EXEC SQL VALUES (SUBSTR(:resume, :di-begin-loc,
              :di-end-loc))
              INTO :di-buffer END-EXEC.
     move "VALUES3" to errloc.
     call "checkerr" using SQLCA errloc.
```

```
* Append the "Department Information" section to the :buffer var
     EXEC SQL VALUES (:buffer || :di-buffer) INTO :buffer
              END-EXEC.
     move "VALUES4" to errloc.
     call "checkerr" using SQLCA errloc.

     go to End-Fetch-Loop.

 NULL-lob-indicated.
     display "NULL LOB indicated".

 End-Fetch-Loop. exit.

 End-Prog.
            stop run.
```

예: LOB 표현식 평가 연기

목표 목적지로의 LOB 표현식 지정이 될 때까지 LOB 값 바이트에는 이동이 없습니다. 이는 문자열 함수 및 연산자와 함께 사용되는 LOB 값 위치 지정자가 지정될 때까지 표현식이 연기되는 표현식을 작성할 수 있음을 의미합니다. 이를 LOB 표현식의 *평가 연기*라고 합니다.

이 예에서는 EMP_RESUME 재개 테이블 내에서 특정 재개(empno = '000130')가 탐색됩니다. 재개의 부서 정보 절은 재개 끝부분에 복사, 잘라내기 및 추가됩니다. 그런 후 이 새로운 재개는 EMP_RESUME 테이블에 삽입됩니다. 이 테이블의 원래 재개는 변경되지 않은 상태로 남아 있습니다.

위치 지정자는 원래 재개로부터 바이트를 실제 이동시키거나 복사하지 않고 새로운 재개의 검토 및 어셈블리를 허용합니다. 바이트의 이동은 최종 할당될 때까지 즉, INSERT문 — 및 서버에서만 발행하지 않습니다.

평가 연기는 DB2에게 LOB I/O 성능을 향상시킬 기회를 제공합니다. LOB 기능 최적화 알고리즘이 LOB 표현식을 대체 표현식으로 변환하려 하기 때문에 발생합니다. 이러한 대체 표현식은 동일한 결과를 산출하지만 보다 적은 수의 디스크 I/O가 필요할 수도 있습니다.

간단히 말해서 LOB 위치 지정자는 수많은 프로그래밍 시나리오에 이상적입니다.

1. 보다 큰 LOB에서 작은 부분만 클라이언트 프로그램으로 이동시킬 때.
2. 전체 LOB가 응용프로그램의 메모리와 맞지 않을 때.
3. 프로그램에 LOB 표현식으로부터의 임시 LOB 값이 필요하지만 결과를 저장할 필요가 없을 때.
4. (LOB 표현식의 평가를 연기하여) 성능이 중요한 경우.

샘플 LOBEVAL 프로그램 작업 방법

1. **호스트 변수 정의.** BEGIN DECLARE SECTION 및 END DECLARE SECTION문은 호스트 변수 선언을 분리합니다. 호스트 변수는 SQL문에서 참조될 때 콜론(:)으로 시작됩니다. CLOB LOCATOR 호스트 변수가 선언됩니다.
2. **LOB 값을 LOCATOR 호스트 변수로 페치.** CURSOR 및 FETCH 루틴을 사용하여 데이터베이스에서의 LOB 필드 위치를 호스트 변수 위치 지정자로 확보합니다.
3. **LOB 데이터는 LOCATORS 사용을 통해 조작됩니다.** 다음 다섯개의 SQL 문은 LOB 필드에 포함된 실제 데이터를 이동시키지 않고 LOB 데이터를 조작합니다. 이는 LOB LOCATORS 사용을 통해 수행됩니다.
4. **LOB 데이터가 목표 목적지로 이동됩니다.** 목표 목적지로 할당된 LOB의 평가는 이 SQL문까지 지연됩니다. 이 LOB문의 평가가 지연되었습니다.
5. **LOB LOCATORS 비우기.** 이 예에서 사용하는 LOB LOCATORS는 이전에 연결된 값으로부터 위치 지정자를 릴리스하여 비워집니다.

CHECKERR 매크로/함수는 오류 체크 유틸리티입니다. 이 오류 체크 유틸리티의 위치는 사용된 프로그래밍 언어에 따라 달라질 수 있습니다.

C DB2 API를 호출하는 C 프로그램에 대해서는 utilapi.c에서 sqlInfoPrint 함수는 utilapi.h에서 API_SQL_CHECK로서참조됩니다. C embedded SQL 프로그램에 대해서는 utilemb.sqc에서 sqlInfoPrint 함수는 utilemb.h에서 EMB_SQL_CHECK로서 참조됩니다.

COBOL CHECKERR은 checkerr.cbl라고 하는 외부 프로그램입니다.

이 오류 체크 유틸리티에 대한 소스 코드에 대해서는 137 페이지의『예제 프로그램에서 GET ERROR MESSAGE 사용』의 내용을 참조하십시오.

C 예: LOBEVAL.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION; 1
      char userid[9];
      char passwd[19];
      sqlint32       hv_start_deptinfo;
      sqlint32       hv_start_educ;
      sqlint32       hv_return_code;
      SQL TYPE IS CLOB(5K) hv_new_section_buffer;
      SQL TYPE IS CLOB_LOCATOR hv_doc_locator1;
      SQL TYPE IS CLOB_LOCATOR hv_doc_locator2;
      SQL TYPE IS CLOB_LOCATOR hv_doc_locator3;
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: LOBEVAL\n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
  EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else {
      printf ("\nUSAGE: lobeval [userid passwd]\n\n");
      return 1;
   } /* endif */

   /* delete any instance of "A00130" from
     previous executions of this sample */
   EXEC SQL DELETE FROM emp_resume WHERE empno = 'A00130';

   /* Use a single row select to get the document */
   EXEC SQL SELECT resume INTO :hv_doc_locator1 FROM emp_resume
      WHERE empno = '000130' AND resume_format = 'ascii';   2
   EMB_SQL_CHECK("SELECT");

   /* Use the POSSTR function to locate the start of
      sections "Department Information" & "Education" */
   EXEC SQL VALUES (POSSTR(:hv_doc_locator1, 'Department Information'))
      INTO :hv_start_deptinfo; 3
   EMB_SQL_CHECK("VALUES1");

   EXEC SQL VALUES (POSSTR(:hv_doc_locator1, 'Education'))
      INTO :hv_start_educ;
   EMB_SQL_CHECK("VALUES2");

   /* Replace Department Information Section with nothing */
   EXEC SQL VALUES (SUBSTR(:hv_doc_locator1, 1, :hv_start_deptinfo -1)
      || SUBSTR (:hv_doc_locator1, :hv_start_educ))
```

```
      INTO :hv_doc_locator2;
   EMB_SQL_CHECK("VALUES3");

   /* Move Department Information Section into the hv_new_section_buffer */
   EXEC SQL VALUES (SUBSTR(:hv_doc_locator1, :hv_start_deptinfo,
      :hv_start_educ -:hv_start_deptinfo)) INTO :hv_new_section_buffer;
   EMB_SQL_CHECK("VALUES4");

   /* Append our new section to the end (assume it has been filled in)
      Effectively, this just moves the Department Information to the bottom
      of the resume. */
   EXEC SQL VALUES (:hv_doc_locator2 || :hv_new_section_buffer) INTO
      :hv_doc_locator3;
   EMB_SQL_CHECK("VALUES5");

   /* Store this resume section in the table. This is where the LOB value
      bytes really move */
   EXEC SQL INSERT INTO emp_resume VALUES ('A00130', 'ascii',
      :hv_doc_locator3);  4
   EMB_SQL_CHECK("INSERT");

   printf ("LOBEVAL completed\n");

   /* free the locators */  5
   EXEC SQL FREE LOCATOR :hv_doc_locator1, :hv_doc_locator2, : hv_doc_locator3;
   EMB_SQL_CHECK("FREE LOCATOR");

   EXEC SQL CONNECT RESET;
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}
/* end of program : LOBEVAL.SQC */
```

COBOL 예: LOBEVAL.SQB

```
 Identification Division.
 Program-ID. "lobeval".

 Data Division.
 Working-Storage Section.

     copy "sqlenv.cbl".
     copy "sql.cbl".
     copy "sqlca.cbl".

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.  1
 01 userid            pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
   49 passwd-name     pic x(18).
 01 hv-start-deptinfo pic s9(9) comp-5.
 01 hv-start-educ     pic s9(9) comp-5.
 01 hv-return-code    pic s9(9) comp-5.
 01 hv-new-section-buffer USAGE IS SQL TYPE IS CLOB(5K).
 01 hv-doc-locator1   USAGE IS SQL TYPE IS CLOB-LOCATOR.
 01 hv-doc-locator2   USAGE IS SQL TYPE IS CLOB-LOCATOR.
 01 hv-doc-locator3   USAGE IS SQL TYPE IS CLOB-LOCATOR.
     EXEC SQL END DECLARE SECTION END-EXEC.

 77 errloc          pic x(80).

 Procedure Division.
 Main Section.
     display "Sample COBOL program: LOBEVAL".

* Get database connection information.
     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR
* format with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".
```

```
     EXEC SQL CONNECT TO sample USER :userid USING :passwd
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

* Delete any instance of "A00130" from previous executions
     EXEC SQL DELETE FROM emp_resume
              WHERE empno = 'A00130' END-EXEC.

* use a single row select to get the document
     EXEC SQL SELECT resume INTO :hv-doc-locator1   2
              FROM emp_resume
              WHERE empno = '000130'
              AND resume_format = 'ascii' END-EXEC.
     move "SELECT" to errloc.
     call "checkerr" using SQLCA errloc.

* use the POSSTR function to locate the start of sections
* "Department Information" & "Education"
     EXEC SQL VALUES (POSSTR(:hv-doc-locator1,
              'Department Information'))
              INTO :hv-start-deptinfo END-EXEC.  3
     move "VALUES1" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL VALUES (POSSTR(:hv-doc-locator1,
              'Education')) INTO :hv-start-educ END-EXEC.
     move "VALUES2" to errloc.
     call "checkerr" using SQLCA errloc.

* replace Department Information section with nothing
     EXEC SQL VALUES (SUBSTR(:hv-doc-locator1, 1,
              :hv-start-deptinfo - 1) ||
              SUBSTR(:hv-doc-locator1, :hv-start-educ))
              INTO :hv-doc-locator2 END-EXEC.
     move "VALUES3" to errloc.
     call "checkerr" using SQLCA errloc.

* move Department Information section into hv-new-section-buffer
     EXEC SQL VALUES (SUBSTR(:hv-doc-locator1,
              :hv-start-deptinfo,
              :hv-start-educ - :hv-start-deptinfo))
              INTO :hv-new-section-buffer END-EXEC.
     move "VALUES4" to errloc.
     call "checkerr" using SQLCA errloc.

* Append the new section to the end (assume it has been filled)
* Effectively, this just moves the Dept Info to the bottom of
```

```
* the resume.
     EXEC SQL VALUES (:hv-doc-locator2 ||
              :hv-new-section-buffer)
              INTO :hv-doc-locator3 END-EXEC.
     move "VALUES5" to errloc.
     call "checkerr" using SQLCA errloc.

* Store this resume in the table.
* This is where the LOB value bytes really move.
     EXEC SQL INSERT INTO emp_resume   4
              VALUES ('A00130', 'ascii', :hv-doc-locator3)
              END-EXEC.
     move "INSERT" to errloc.
     call "checkerr" using SQLCA errloc.

     display "LOBEVAL completed".

     EXEC SQL FREE LOCATOR :hv-doc-locator1, :hv-doc-locator2,  5
              :hv-doc-locator3 END-EXEC.
     move "FREE LOCATOR" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL CONNECT RESET END-EXEC.
     move "CONNECT RESET" to errloc.
     call "checkerr" using SQLCA errloc.

 End-Prog.
     stop run.
```

표시기 변수 및 LOB 위치 지정자

응용프로그램에 있는 일반 호스트 변수의 경우, 호스트 변수에 널(NULL) 값을 선택할 때 그 값이 널(NULL)임을 의미하는 표시기 변수에 음수 값이 할당됩니다. 그러나 LOB 위치 지정자의 경우 표시기 변수의 의미는 약간씩 다릅니다. 위치 지정자 호스트 변수 자체는 결코 널(NULL)일 수 없으므로, 음수 표시기 변수 값은 LOB 위치 지정자에 의해 표시되는 LOB 값이 널(NULL)임을 나타냅니다. 널(NULL) 정보는 표시기 변수 값을 사용하여 클라이언트에 지역 상태로 남아 있습니다. 서버는 유효한 위치 지정자로 널(NULL) 값을 추적하지 않습니다.

LOB 파일 참조 변수

파일 참조 변수는 메모리 버퍼가 아닌 클라이언트 파일로 데이터를 전송하는 점을 제외하고, 호스트 변수와 유사합니다. 파일 참조 변수는 LOB 위치 지정자가 LOB 위치 지정자(을 포함하는 것이 아닌)를 표시하는 것과 마찬가지로 파일(을 포함되는 것이 아니라)을 표시합니다. 데이터베이스 조회, 갱신 및 삽입은 파일 참조 변수를 사용하여 단일 LOB 값을 저장하거나 검색합니다.

초대형 오브젝트(LOB)의 경우 파일은 본래의 컨테이너입니다. 사실 대부분의 LOB는 서버에 있는 데이터베이스로 이동되기 전에 클라이언트에 있는 파일에 저장된 데이터로서 시작합니다. 파일 참조 변수를 사용함으로써 LOB 데이터를 이동시키는 것이 쉬워집니다. 프로그램은 파일 참조 변수를 사용하여 클라이언트 파일로부터 데이터베이스 엔진으로 직접 LOB 데이터를 전송합니다. 클라이언트 응용프로그램은 LOB 데이터 이동을 실행하기 위해 (크기 제한이 있는) 호스트 변수를 사용하여 파일을 읽기 및 쓰기 위해 유틸리티 루틴을 작성할 필요가 없습니다.

주: 파일 참조 변수에 의해 참조되는 파일은 프로그램이 수행되는 시스템(에 반드시 상주할 필요는 없지만)으로부터 액세스할 수 있어야 합니다. 저장 프로시듀어의 경우 이것은 서버일 수 있습니다.

파일 참조 변수의 데이터 유형은 BLOB, CLOB 또는 DBCLOB입니다. 데이터 유형은 데이터 소스(입력) 또는 데이터 목표(출력)로서 사용됩니다. 파일 참조 변수에는 파일의 전체 경로 이름 또는 상대적 파일 이름이 있을 수 있습니다.(후자를 추천합니다.) 파일 이름 길이는 응용프로그램 내에서 지정됩니다. 입력 중에 파

일 참조 변수의 데이터 길이 부분은 사용되지 않습니다. 출력 중에 데이터 길이는 응용프로그램 요청자 코드에 의해 파일에 쓰여진 새로운 데이터 길이로 설정됩니다.

파일 참조 변수 사용시 입력 및 출력 모두에 대해 다른 옵션이 있습니다. 파일 참조 변수 구조에 `file_option` 필드를 설정하여 파일에 대한 조치를 선택해야 합니다. 입력 및 출력 값 모두를 다루는 필드로의 지정에 대한 선택사항들은 아래에 나와 있습니다.

입력 파일 참조 변수를 사용할 때의 옵션 및 (C용으로 표시되는) 값은 다음과 같습니다.

- **SQL_FILE_READ**(일반 파일) - 열고, 읽고, 닫을 수 있는 파일입니다. DB2는 파일을 열 때 파일에 있는 데이터 길이(바이트 단위)를 결정합니다. 그런 후 DB2는 파일 참조 변수 구조의 `data_length` 필드를 통해 길이를 리턴합니다.(COBOL의 값은 SQL-FILE-READ이고, FORTRAN 값은 sql_file_read입니다.)

출력 파일 참조 변수를 사용할 때의 값 및 옵션은 다음과 같습니다.

- **SQL_FILE_CREATE**(새 파일) - 이 옵션은 새로운 파일을 작성합니다. 파일이 이미 존재하는 경우 오류 메시지가 리턴됩니다.(COBOL에 대한 값은 SQL-FILE-CREATE이고, FORTRAN에 대한 값은 sql_file_create입니다.)
- **SQL_FILE_OVERWRITE**(겹쳐쓰기 파일) - 이 옵션은 파일이 없는 경우 새로운 파일을 작성합니다. 이미 파일이 있는 경우 새로운 데이터는 파일에 있던 데이터를 겹쳐씁니다.(COBOL에 대한 값은 SQL-FILE-OVERWRITE이고, FORTRAN에 대한 값은 sql_file_overwrite입니다.)
- **SQL_FILE_APPEND**(추가 파일) - 이 옵션은 파일이 있는 경우 그 파일에 출력을 추가합니다. 그렇지 않으면 새로운 파일이 작성됩니다.(COBOL에 대한 값은 SQL-FILE-APPEND이고, FORTRAN에 대한 값은 sql_file_append입니다.)

주:

1. Extended UNIX Code(EUC) 환경에서, DBCLOB 파일 참조 변수가 가리키는 파일에는 그래픽 컬럼에서 저장영역에 적절한 유효 EUC 문자가 포함되어

있고, UCS-2 문자는 들어 있지 않다고 간주됩니다. EUC 환경의 DBCLOB 파일에 대한 세부사항은 586 페이지의 『DBCLOB 파일에 대한 고려사항』의 내용을 참조하십시오.

2. LOB 파일 참조 변수가 OPEN문에서 사용될 경우, LOB 파일 참조 변수와 연관된 파일은 커서가 닫힐 때까지 삭제해서는 안 됩니다.

파일 참조 변수에 대해서는 *SQL 참조서*에서 자세한 내용을 참조하십시오.

예: 파일로 문서 발췌

이 프로그램 예는 CLOB 요소가 테이블로부터 외부 파일로 수신될 수 있는 방법을 보여 줍니다.

샘플 LOBFILE 프로그램 작업 방법

1. **호스트 변수 정의.** BEGIN DECLARE SECTION 및 END DECLARE SECTION문은 호스트 변수 선언을 분리합니다. 호스트 변수는 SQL문에서 참조될 때 콜론(:)으로 시작됩니다. CLOB FILE REFERENCE 호스트 변수가 선언됩니다.
2. **CLOB FILE REFERENCE 호스트 변수가 설정됩니다.** FILE REFERENCE의 속성이 설정됩니다. 선언된 전체 경로가 없는 파일 이름이 기본값으로 현재 작업 디렉토리에 놓여 집니다.
3. **CLOB FILE REFERENCE 호스트 변수로 선택.** 재개 필드의 데이터는 호스트 변수에 의해 참조되는 파일 이름으로 선택됩니다.

CHECKERR 매크로/함수는 오류 체크 유틸리티입니다. 이 오류 체크 유틸리티의 위치는 사용된 프로그래밍 언어에 따라 달라질 수 있습니다.

C
: DB2 API를 호출하는 C 프로그램에 대해서는 utilapi.c에서 sqlInfoPrint 함수는 utilapi.h에서 API_SQL_CHECK로서참조됩니다. C embedded SQL 프로그램에 대해서는 utilemb.sqc에서 sqlInfoPrint 함수는 utilemb.h에서 EMB_SQL_CHECK로서 참조됩니다.

COBOL
: CHECKERR은 checkerr.cbl라고 하는 외부 프로그램입니다.

이 오류 체크 유틸리티에 대한 소스 코드에 대해서는 137 페이지의 『예제 프로그램에서 GET ERROR MESSAGE 사용』의 내용을 참조하십시오.

C 예: LOBFILE.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sql.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION;  1
      SQL TYPE IS CLOB_FILE resume;
      short lobind;
      char userid[9];
      char passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: LOBFILE\n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
  EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else {
      printf ("\nUSAGE: lobfile [userid passwd]\n\n");
      return 1;
   } /* endif */

   strcpy (resume.name, "RESUME.TXT");  2
   resume.name_length = strlen("RESUME.TXT");
   resume.file_options = SQL_FILE_OVERWRITE;

   EXEC SQL SELECT resume INTO :resume :lobind FROM emp_resume  3
      WHERE resume_format='ascii' AND empno='000130';

   if (lobind < 0) {
```

```
      printf ("NULL LOB indicated \n");
   } else {
      printf ("Resume for EMPNO 000130 is in file : RESUME.TXT\n");
   } /* endif */

   EXEC SQL CONNECT RESET;
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}
/* end of program : LOBFILE.SQC */
```

COBOL 예: LOBFILE.SQB

```
 Identification Division.
 Program-ID. "lobfile".

 Data Division.
 Working-Storage Section.

     copy "sqlenv.cbl".
     copy "sql.cbl".
     copy "sqlca.cbl".

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.  1
 01 userid             pic x(8).
 01 passwd.
   49 passwd-length    pic s9(4) comp-5 value 0.
   49 passwd-name      pic x(18).
 01 resume             USAGE IS SQL TYPE IS CLOB-FILE.
 01 lobind             pic s9(4) comp-5.
     EXEC SQL END DECLARE SECTION END-EXEC.

 77 errloc           pic x(80).

 Procedure Division.
 Main Section.
     display "Sample COBOL program: LOBFILE".

* Get database connection information.
     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR
* format with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.
```

```
    move "RESUME.TXT" to resume-NAME.  2
    move 10 to resume-NAME-LENGTH.
    move SQL-FILE-OVERWRITE to resume-FILE-OPTIONS.

    EXEC SQL SELECT resume INTO :resume :lobind   3
             FROM emp_resume
             WHERE resume_format = 'ascii'
             AND empno = '000130' END-EXEC.
    if lobind less than 0 go to NULL-LOB-indicated.

    display "Resume for EMPNO 000130 is in file : RESUME.TXT".
    go to End-Main.

NULL-LOB-indicated.
    display "NULL LOB indicated".

End-Main.
    EXEC SQL CONNECT RESET END-EXEC.
    move "CONNECT RESET" to errloc.
    call "checkerr" using SQLCA errloc.
End-Prog.
           stop run.
```

예: CLOB 컬럼에 데이터 삽입

다음 C 프로그램 세그먼트의 경로 설명에서:

- userid는 사용자 중 하나에 대한 디렉토리를 나타냅니다.
- dirname은 『userid』의 서브디렉토리 이름을 나타냅니다.
- filnam.1은 테이블에 삽입하고자 하는 사용자 문서의 이름이 될 수 있습니다.
- clobtab는 데이터 유형이 CLOB인 테이블의 이름입니다.

다음 예에서는 :hv_text_file에 의해 참조되는 일반 파일로부터 CLOB 컬럼으로 데이터를 삽입하는 방법을 보여줍니다.(예에 사용된 경로 이름은 UNIX 시스템용입니다.)

```
strcpy(hv_text_file.name, "/u/userid/dirname/filnam.1");
hv_text_file.name_length = strlen("/u/userid/dirname/filnam.1");
hv_text_file.file_options = SQL_FILE_READ; /* this is a 'regular' file */

EXEC SQL INSERT INTO CLOBTAB
  VALUES(:hv_text_file);
```

제14장 사용자 정의 함수(UDFs)와 메소드

함수와 메소드는 무엇입니까? 409
왜 함수와 메소드를 사용합니까? 410
UDF 및 Method 개념 413
메소드 및 함수 구현. 416
메소드 및 함수 작성. 416
메소드 및 함수 등록. 416
UDF 및 메소드 등록의 예. 417
  예: 지수화 417
  예: 문자열 검색 418
  예: BLOB 문자열 검색. 419
  예: UDT에서의 문자열 검색 419
  예: UDT 매개변수가 있는 외부 함수 . . 420
  예: UDT의 AVG 421
  예: 카운팅 421
  예: OLE 자동화 오브젝트에서의 카운팅 422
  예: 문서 ID를 리턴하는 테이블 함수 . . 422
메소드 및 함수 사용. 424
  함수 참조 424
  함수 호출 예 425
  함수의 매개변수 표시문자 사용 426
  규정화된 함수 참조 사용 426
  규정화되지 않은 함수 참조 사용. 427
  함수 참조 요약 428

함수와 메소드는 무엇입니까?

*사용자 정의 함수(UDF)*는 사용자 자신의 확장자를 SQL에 쓸 수 있는 메카니즘입니다. DB2와 함께 제공되는 내장 함수는 유용한 함수 세트이지만, 사용자의 모든 요건을 만족시키지는 못합니다. DB2와 함께 제공되는 완전한 함수 목록에 대해 *SQL 참조서*의 "지원되는 함수" 테이블을 참조하십시오.

UDF와 같은 *메소드*는 SQL 오브젝트의 동작을 정의하여 SQL에 자체 확장을 작성할 수 있게 합니다. 그러나 UDF와는 달리 메소드를 테이블의 컬럼으로 저장된 구조화된 유형과만 연관시킬 수 있습니다.

다음과 같은 이유로 SQL을 확장시켜야 할 경우가 있습니다.

- **사용자 정의.**

  응용프로그램별 함수가 DB2에 존재하지 않습니다. 함수가 단순 변형인지, 사소한 계산인지 또는 복잡한 다중 변이 분석인지에 관계없이, UDF를 사용하여 작업을 수행할 수 있습니다.

- **융통성.**

  DB2 내장 함수는 응용프로그램에 포함시키고자 하는 변이를 허용하지 않습니다.

© Copyright IBM Corp. 1993, 2000

- **표준화**.

  사용자 사이트의 많은 프로그램들은 동일한 기본 함수 세트를 구현하지만, 모든 구현에는 사소한 차이가 있습니다. 따라서, 수신되는 결과의 일치 여부가 불확실할 수 있습니다. UDF에서 이 함수를 한 번 제대로 구현하면, 이 모든 프로그램들이 SQL에서 직접 동일한 구현을 사용하여 일치된 결과를 제공할 수 있습니다.

- **오브젝트 관계형 지원**.

  303 페이지의 『제11장 사용자 정의 구별 유형』 및 315 페이지의 『제12장 복잡한 오브젝트 사용: 사용자 정의 구조화 유형』에서 설명된 바와 같이 구별 유형과 구조화된 유형은 DB2의 기능을 확장하고 안전성을 증가시키는 데 매우 유용합니다. 컬럼에 저장된 구조화된 유형의 동작을 정의할 수 있는 메소드를 작성할 수 있습니다. 구별 유형에서 작동하는 함수를 작성할 수도 있습니다.

왜 함수와 메소드를 사용합니까?

DB2 응용프로그램을 작성하는 데 있어서 사용자는 원하는 조치 또는 작업을 구현할 때 선택을 할 수 있습니다.

- UDF로서
- 메소드로서
- 응용프로그램의 서브루틴으로서

응용프로그램에서 서브루틴으로서 새로운 조작을 구현하는 것이더 쉬워 보여도 UDF 사용을 고려해야 할 충분한 이유가 있습니다.

- **재사용**.

  새로운 작업이 사용자 사이트에서 다른 사용자 또는 프로그램이 장점을 취할 수 있는 것이면 UDF 및 메소드가 이를 재사용하도록 도울 수 있습니다. 또한, 표현식을 데이터베이스 사용자가 사용할 수 있는 SQL에서 작업을 직접호출할 수 있습니다. UDF에 대해 데이터베이스는 함수 인수의 여러 데이터 유형을 자동으로 승격(예:DECIMAL에서 DOUBLE로)시킵니다. 이를 통해 서로 다른지만 호화 가능한 데이터 유형에 조작을 적용시킬 수 있습니다.

  새 조작을 쉽게 서브루틴으로 구현하고 다른 사용자가 자신의 프로그램에서 사용할 수 있게 만들어 DB2에서 함수를 정의하지 않아도 되게 할 수 있습니다.

다른 모든 응용프로그램 개발자에게 알려야 하고효율적으로 사용할 수 있도록 서브루틴을 패키지로 만들어야 합니다. 그러나, 일반적으로 명령행 처리기(CLP)를 사용하여 데이터베이스에 액세스하는 사람들과 마찬가지로 대화식 사용자를 무시합니다. CLP 사용자는 데이터베이스에 있는 UDF 또는 메소드가 아닌 한 사용자 함수를 사용할 수 없습니다. 이는 또한 재컴파일될 수 없는 SQL(예:Lotus Approach)을 사용하는 기타 도구에도 적용됩니다.

- 성능.

  응용프로그램 대신 직접 데이터베이스 엔진에서 UDF나 메소드를 호출하면 특히 데이터를 더 심도 있게 처리할 수 있도록 규정하는 경우 상당한 성능 향상을 얻을 수 있습니다. `SELECTION_CRITERIA()` 함수로서 표현할 수 있는 일부 선택 기준을 충족시킬 수 있다면, 일부 데이터를 처리하고자 하는 간단한 시나리오를 고려하십시오. 사용자 응용프로그램은 다음과 같은 select문을 발행했을 수 있습니다.

```
SELECT A,B,C FROM T
```

  각 행을 받을 때 데이터를 심화 처리하는 데 관심이 있는지 결정하기 위해 데이터에 대해 `SELECTION_CRITERIA`를 수행합니다. 여기서 각 행의 테이블 T는 다시 응용프로그램으로 전달되어야 합니다. 그러나 `SELECTION_CRITERIA()`가 UDF로서 구현되면 사용자 응용프로그램은 다음 명령문을 발행할 수 있습니다.

```
SELECT A,B,C FROM T WHERE SELECTION_CRITERIA(A,B) = 1
```

  이런 경우, 응용프로그램과 데이터베이스 사이의 인터페이스를 통해 관심 있는 행들만 전달됩니다. 대형 테이블의 경우, 또는 `SELECTION_CRITERIA`가 주요 필터링을 제공하는 경우 성능 향상이 매우 중요할 것입니다.

  UDF가 성능 향상을 제공할 수 있는 또다른 경우는 대형 오브젝트(LOB)를 다룰 때입니다. LOB 유형의 값에서 어떤 정보를 추출하는 것이 목적인 함수인 경우, 데이터베이스 서버에서 직접 이 정보를 추출하여 추출된 값만 다시 응용프로그램으로 전달할 수 있습니다. 이는 전체 LOB 값을 다시 응용프로그램으로 전달한 후 추출을 수행하는 것보다 훨씬 효율적입니다. UDF로서 이 함수를 패키지화하는 성능 값은 상황에 따라 매우 클 수 있습니다.(LOB 위치 지정자를 사용하여 LOB의 일부도 추출할 수 있습니다. 유사한 시나리오의 예는 394 페이지의 『예: LOB 표현식 평가 연기』의 내용을 참조하십시오.)

이 외에도, CREATE FUNCTION문의 RETURNS TABLE절을 사용하여 *테이블 함수*라고 하는 UDF를 호출할 수 있습니다. 테이블 함수로 (비관계형 데이터 저장을 포함하여) DB2 데이터베이스 외부에 상주하는 데이터에 대해 SQL의 관계형 작업과 성능을 매우 효율적으로 사용할 수 있습니다. 테이블 함수는 인수로서 개별적인 스칼라 값의 다른 유형 및 의미를 취해, 이를 호출하는 SQL 문으로 테이블을 리턴합니다. 원하지 않는 행 또는 컬럼은 제거시키고 관심있는 데이터만 생성하는 테이블 함수를 작성할 수 있습니다. 사용할 수 있는 경우에 대한 규칙을 포함하여 테이블 함수에 대한 세부사항은 *SQL 참조서*의 내용을 참조하십시오.

테이블을 리턴하는 메소드를 작성할 수 없습니다.

- **구별 유형의 동작**.

  UDF를 사용하여 *구별 유형*이라고도 하는 사용자 정의 구별 유형(UDT)의 조작을 구현할 수 있습니다. UDT에 대한 세부사항은 303 페이지의 『제11장 사용자 정의 구별 유형』의 내용을 참조하십시오. UDT에 대한 세부사항 및 그 안에서 논의된 *유형변환 기능*에 대한 주요 개념은 *SQL 참조서*의 내용을 참조하십시오. 구별 유형을 작성할 때 구별 유형과 소스 유형 사이에 유형변환(cast) 함수가 자동으로 제공되며, 소스 유형에 따라 =, >, < 등의 비교 연산자가 제공됩니다. 사용자 스스로 추가 조작을 제공해야 합니다. 구별 유형의 모든 사용자가 쉽게 액세스할 수 있는 데이터베이스에 구별 유형의 조작을 보유하는 것이 바람직하므로, UDF를 구현 메카니즘으로서 사용할 수 있습니다.

  예를 들어, 1MB BLOB를 통해 정의된 BOAT 구별 유형이 있다고 가정하십시오. BLOB에는 다양한 항해 스펙 및 드로잉이 들어 있습니다. 보트 *크기*를 비교하고 싶지만, BLOB 소스 유형을 통해 정의된 구별 유형을 가지고 자동으로 생성된 비교 작업을 얻을 수 없습니다. 선택한 측정 단위를 기초로 하여 한 보트가 다른 보트보다 큰지 결정하는 BOAT_COMPARE 함수를 구현할 수 있습니다. 치환, 전체 길이, 톤 중량 또는 BOAT 오브젝트에 기초한 다른 계산이 사용될 수 있습니다. 다음과 같은 BOAT_COMPARE 함수를 작성합니다.

  ```
  CREATE FUNCTION BOAT_COMPARE (BOAT, BOAT) RETURNS INTEGER ...
  ```

  첫번째 BOAT가 더 클 경우 사용자 함수가 1을 리턴하고, 두 번째가 더 클 경우 2를, 두 *크기*가 같을 경우 0을 리턴하면, SQL 코드에서 이 함수를 사용하여 보트를 비교할 수 있습니다. 다음 테이블을 작성해 봅니다.

```
CREATE TABLE BOATS_INVENTORY (
  BOAT_ID        CHAR(5),
  BOAT_TYPE      VARCHAR(25),
  DESIGNER       VARCHAR(40),
  OWNER          VARCHAR(40),
  DESIGN_DATE    DATE,
  SPEC           BOAT,
  ...                  )

CREATE TABLE MY_BOATS (
  BOAT_ID        CHAR(5),
  BOAT_TYPE      VARCHAR(25),
  DESIGNER       VARCHAR(40),
  DESIGN_DATE    DATE,
  ACQUIRE_DATE   DATE,
  ACQUIRE_PRICE  CANADIAN_DOLLAR,
  CURR_APPRAISL  CANADIAN_DOLLAR,
  SPEC           BOAT,
  ...                  )
```

다음과 같은 SQL SELECT문을 실행할 수 있습니다.

```
SELECT INV.BOAT_ID, INV.BOAT_TYPE, INV.DESIGNER,
  INV.OWNER, INV.DESIGN_DATE
  FROM BOATS_INVENTORY INV, MY_BOATS MY
  WHERE MY.BOAT_ID = '19GCC'
  AND BOAT_COMPARE(INV.SPEC, MY.SPEC) = 1
```

이 간단한 예에서는 BOATS_INVENTORY로부터 MY_BOATS의 특정 보트보다 큰 모든 보트를 리턴합니다. 데이터베이스 서버에서 비교가 발생하므로, 이 예는 관심있는 행만 다시 응용프로그램으로 전달합니다. 사실 데이터 유형 BOAT 값을 전달할 여지가 완전히 없어집니다. 이는 BOAT가 1MB의 BLOB 데이터 유형에 기초하므로 저장영역 및 성능이 향상됩니다.

UDF 및 Method 개념

다음은 UDF 및 메소드를 코드화하기 전에 알아야 할 주요 개념입니다.

- 함수의 전체 이름

  함수의 전체 이름은 <schema-name>.<function-name>입니다. 함수를 참조할 수 있는 경우 이 전체 이름을 사용할 수 있습니다. 예를 들어 다음과 같습니다.

  ```
  SLICKO.BOAT_COMPARE    SMITH.FOO    SYSIBM.SUBSTR    SYSFUN.FLOOR
  ```

그러나, DB2가 사용자가 참조하고 있는 함수를 식별해야할 경우 <schema-name>.도 생략할 수 있습니다. 예를 들어 다음과 같습니다.

```
BOAT_COMPARE    FOO    SUBSTR    FLOOR
```

- 함수 경로

  *함수 경로*의 개념은 schema-name을 사용하지 않을 때 발생하는 *규정화되지 않*은 참조의 DB2 분석에 중요합니다. 함수를 참조하는 DDL문에서의 함수 경로 사용에 대해서는 *SQL 참조서*의 내용을 참조하십시오. 함수 경로는 스키마 이름의 순서화된 목록입니다. 함수 경로는 UDT 뿐만 아니라 UDF 및 메소드에 대한 규정화되지 않은 함수 참조 분석용 스키마 세트도 제공합니다. 함수 참조가 경로의 여러 스키마에 있는 함수와 일치할 경우 경로의 스키마 순서를 사용하여 이러한 일치를 분석합니다. 함수 경로는 정적 SQL에 대한 사전처리 컴파일 및 바인드 명령에 대한 FUNCPATH 옵션을 이용하여 설정됩니다. 함수 경로는 동적 SQL에 대한 SET CURRENT FUNCTION PATH문에 의해 설정됩니다. 함수 경로의 기본값은 다음과 같습니다.

  ```
  "SYSIBM","SYSFUN","<ID>"
  ```

  이것은 정적 SQL과 동적 SQL 모두에 적용됩니다. 여기서 <ID>는 현재 명령문 권한 부여 ID를 표시합니다.

- 오버로드된 함수 이름

  함수 이름은 *오버로드*될 수 있습니다. 이는 같은 스키마에서도 여러 개의 함수가 같은 이름을 가질 수 있음을 의미합니다. 그러나 두 함수는 같은 *신호 (signature)*를 가질 수 없습니다. 정의된 순서로 모든 함수 매개변수의 정의된 데이터 유형과 연결된 규정화된 함수 이름으로 정의될 수 있습니다. 오버로드된 함수의 예는 419 페이지의 『예: BLOB 문자열 검색』의 내용을 참조하십시오.

- 함수 선택 알고리즘

  규정화된 참조이든 규정화되지 않은 참조이든, 모든 함수 참조에 *가장 잘 맞는* 것을 선택하기 위해 함수 경로 및 오버로드를 고려하는 *함수 선택 알고리즘*입니다. 내장 함수 및 SYSFUN 스키마에 있는 (IBM 제공) 함수에 대한 참조도 함수 선택 알고리즘을 통해 처리됩니다.

- 함수 유형

각 사용자 정의 함수(UDF)는 *스칼라*, *컬럼* 또는 *테이블* 함수로 분류됩니다. *스칼라 함수*는 호출될 때마다 단일 값 응답을 리턴합니다. 예를 들어, 내장 함수 SUBSTR()은 스칼라 함수입니다. 스칼라 UDF 및 메소드는 (C와 같은프로그래밍 언어로 코드화된) 외부 또는 (기존 함수의 구현을 사용하는) 소스가 될 수 있습니다.

*컬럼 함수*는 유사 함수 세트(데이터 컬럼)를 받아 이 값 세트로부터 단일 값 응답을 리턴합니다. DB2에서는 이를 *총계 함수*라고도 합니다. 컬럼 함수의 예로는 내장 함수 AVG()가 있습니다. 외부 컬럼 UDF는 DB2로 정의될 수 없지만, 내장 컬럼 함수 중 하나에 소스가 있는 컬럼 UDF를 정의할 수 있습니다. 이는 구별 유형에 유용합니다. 예를 들어, INTEGER 구별 유형으로 정의된 SHOESIZE가 존재하는 경우, 기존의 내장 컬럼 함수, AVG(INTEGER)에 소스가 있는 컬럼 함수로서 UDF, AVG(SHOESIZE)를 정의할 수 있습니다.

*테이블 함수*는 테이블을 참조하는 SQL문으로 테이블을 리턴합니다. 테이블 함수는 SELECT문의 FROM절에서만 참조될 수 있습니다. 그러한 함수를 사용하여 비-DB2 데이터에 SQL 언어를 적용시키거나, 이런 데이터를 캡쳐해서 DB2 테이블에 둘 수 있습니다. 예를 들어, 이는 비-DB2 데이터로 구성된 파일을 테이블로 동적 변환시키거나 WWW 또는 운영 체제에서 데이터를 검색하고 데이터를 테이블로 리턴할 수 있습니다. 테이블 함수는 외부 함수만 될 수 있습니다.

함수 선택 알고리즘, SET CURRENT FUNCTION PATH문 및 함수 경로의 개념은 *SQL 참조서*에 상세하게 논의되어 있습니다. FUNCPATH 사전 처리 컴파일 및 바인드 옵션에 대해서는 *Command Reference*에 상세히 논의되어 있습니다.

연합 시스템에서 UDF와 메소드 및 내장 함수를 데이터 소스 함수로 맵핑하는 개념에 대해서는 *SQL 참조서*의 내용을 참조하십시오. 이러한 맵핑 작성 방법에 대해서는 653 페이지의 『데이터 소스 함수 호출』의 내용을 참조하십시오.

메소드 및 함수 구현

외부 UDF 및 메소드 구현 프로세스에는 다음과 같은 단계가 필요합니다.

1. UDF 또는 메소드 작성
2. UDF 또는 메소드 컴파일
3. UDF 또는 메소드 링크
4. UDF 또는 메소드 디버깅
5. DB2에서 UDF 또는 메소드 등록

이 단계가 성공적으로 완료되면 UDF 또는 메소드가 CREATE VIEW와 같은 DML 또는 DDL문에서 사용할 준비가 된 것입니다. UDF 및 메소드 작성 및 정의 단계에 대해서는다음 절에 나와 있고, 그 다음에는 UDF 및 메소드 사용에 대한 논의가 나옵니다. UDF 및 메소드 컴파일 및 링크에 대해서는 *응용프로그램 빌드 안내서*의 내용을 참조하십시오. UDF 또는 메소드 디버깅에 대해서는 535 페이지의 『UDF 디버깅』의 내용을 참조하십시오.

메소드 및 함수 작성

433 페이지의 『제15장 사용자 정의 함수(UDF)』에서 UDF 및 메소드를 작성하는 방법에 대한 세부사항을 볼 수있습니다. 여기에는 DB2와 UDF간의 인터페이스 또는 메소드, 코딩 고려사항, 코딩 예 그리고 디버깅 정보에 대한 세부사항이 들어 있습니다. UDF 및 메소드 컴파일과 링크에 관련된 작업에 대한 자세한 내용은 *응용프로그램 빌드 안내서*을 참조하십시오.

메소드 및 함수 등록

실제 코드를 작성 및 완전 테스트한 후에 UDF 또는 메소드를 DB2에 레지스터해야 합니다. 실제로 작성하기 전에 UDF 또는 메소드를 정의하는 것이 가능합니다. 그러나, UDF 또는 메소드수행시 문제를 피하려면 레지스터하기 전에 광범위하게 작성 및 테스트하도록 해야 합니다. UDF 또는 메소드 테스트에 대해서는 535 페이지의 『UDF 디버깅』의 내용을 참조하십시오.

UDF를 DB2에 정의(또는 등록)하려면 CREATE FUNCTION문을 사용하십시오. DB2와 함께 메소드를 등록하려면 CREATE TYPE 또는 ALTER TYPE문을 사

용하여 구조화된 유형의 메소드를 정의한 후 CREATE METHOD문을 사용하여 메소드 본문을 메소드 스펙과 연관시킵니다. *SQL 참조서*에서 이런 명령문의 상세한 설명과 구문을 찾을 수 있습니다.

UDF 및 메소드 등록의 예

다음 예는 UDF 및 메소드가 등록될 수 있는 다양한 전형적인 상황을 보여줍니다.

이 예에서 다음 사항에 유의하십시오.

- 표시의 일관성 및 이해를 돕기 위해 키워드 또는 키워드/값 스펙은 항상 같은 순서로 표시됩니다. 이 CREATE FUNCTION 또는 CREATE METHOD문들 중 하나를 작성하는 데 있어서 함수 이름 및 매개변수 데이터 유형 목록 다음에는 임의의 순서대로 스펙이 나타날 수 있습니다.
- EXTERNAL NAME절의 스펙은 항상 UNIX 플랫폼 DB2에 대해 표시됩니다. UNIX가 아닌 플랫폼에서 이러한 예제를 수행할 경우 변경을 해야 할 수도 있습니다. 예를 들어, 모든 슬래시(/) 문자를 역슬래시(\) 문자로 변환하고 `C:`와 같은 드라이브 이름을 추가함으로써, OS/2 또는 Windows 환경에 유효한 예들이 있습니다. EXTERNAL NAME절에 대한 세부사항은 *SQL 참조서*의 내용을 참조하십시오.

예: 지수화

외부 UDF를 작성하여 부동 소수점 값의 지수화를 수행했고, MATH 스키마에서 이를 등록하려 한다고 가정하십시오. DBADM 권한을 가지고 있다고 가정합니다. 광범위하게 함수를 테스트했고 무결성 표시를 나타내지 않는다는 것을 알고 있으므로, 이를 NOT FENCED로 정의합니다. DBADM 권한이 있으므로 데이터베이스 권한, CREATE_NOT_FENCED를 소유하고 있습니다. 이 권한은 NOT FENCED로서 함수를 정의하는 데 필수적입니다.

```
CREATE FUNCTION MATH.EXPON (DOUBLE, DOUBLE)
  RETURNS DOUBLE
  EXTERNAL NAME '/common/math/exponent'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
```

```
NO SQL
DETERMINISTIC
NO EXTERNAL ACTION
NOT FENCED
```

이 예에서 시스템은 NOT NULL CALL 기본값을 사용합니다. 이는 한 쪽 인수가 널(NULL)인 경우 결과가 널(NULL)이므로 바람직합니다. 스크래치 패드가 필요하지 않고 최종 호출이 필요하지 않으므로 NO SCRATCHPAD 및 NO FINAL CALL 기본값이 사용됩니다. EXPON이 병렬 처리될 수 없는 이유가 없으므로 ALLOW PARALLELISM 기본값이 사용됩니다.

예: 문자열 검색

사용자의 동료 Willie가 UDF를 작성하여 인수로서 전달되어 제공된 short 문자열의 존재를 검색했고, 제공된 CLOB 값 내에서도 인수로서 전달됩니다. UDF는 문자열을 찾을 경우에는 CLOB 내에서의 문자열 위치를 리턴하고, 문자열이 없을 경우에는 제로를 리턴합니다. UDF가 완전히 테스트되지 않았는지 의심되므로 이 함수에 대한 데이터베이스 무결성과 관계가 있으므로 함수를 FENCED로서 정의합니다.

또한, Willie가 함수를 작성하여 FLOAT 결과를 리턴했습니다. SQL에서 사용될 때 항상 INTEGER를 리턴해야 한다고 알고 있다고 가정하십시오. 다음 함수를 작성할 수 있습니다.

```
  CREATE FUNCTION FINDSTRING (CLOB(500K), VARCHAR(200))
    RETURNS INTEGER
    CAST FROM FLOAT
    SPECIFIC "willie_find_feb95"
    EXTERNAL NAME '/u/willie/testfunc/testmod!findstr'
LANGUAGE C
    PARAMETER STYLE DB2SQL
    NO SQL
    DETERMINISTIC
    NO EXTERNAL ACTION
    FENCED
```

UDF 내용이 정말 FLOAT값을 리턴하도록 지정하기 위해 CAST FROM절이 사용되지만, 이 값을 UDF를 사용한 명령문으로 리턴하기 전에 이를 INTEGER로 유형변환하려 합니다. *SQL 참조서*에서 설명한 바와 같이, INTEGER 내장 함수는 사용자 대신 이 유형변환을 수행할 수 있습니다. 또한, 함수에 대해 사용자의

구체적인 이름을 제공하고 나중에 이를 DDL에서 참조하려 합니다(『예: UDT에서의 문자열 검색』 참조). UDF가 널(NULL) 값을 처리하기 위해 작성되지 않았으므로 NOT NULL CALL 기본값을 사용합니다. 스크래치 패드가 없으므로 NO SCRATCHPAD 및 NO FINAL CALL 기본값을 사용합니다. FINDSTRING이 병렬 처리될 수 없는 이유가 없으므로 ALLOW PARALLELISM 기본값이 사용됩니다.

예: BLOB 문자열 검색

이 함수가 CLOB에서 뿐만 아니라 BLOB에서 작동되기를 원하므로, 첫번째 매개변수로서 BLOB를 가져가는 또다른 FINDSTRING을 정의합니다.

```
     CREATE FUNCTION FINDSTRING (BLOB(500K), VARCHAR(200))
       RETURNS INTEGER
       CAST FROM FLOAT
       SPECIFIC "willie_fblob_feb95"
       EXTERNAL NAME '/u/willie/testfunc/testmod!findstr'
LANGUAGE C
       PARAMETER STYLE DB2SQL
       NO SQL
       DETERMINISTIC
       NO EXTERNAL ACTION
       FENCED
```

이 예는 UDF 이름의 오버로드를 보여주고 다중 UDF 및 메소드가 같은 내용을 공유할 수 있음을 보여줍니다. BLOB가 CLOB에 지정될 수 없어도 동일한 소스 코드를 사용할 수 없습니다. DB2와 UDF 사이의 BLOB 및 CLOB에 대한 프로그래밍 인터페이스가 길이가 같고 뒤이어 데이터가 오므로 위 예에서 프로그래밍 문제가 없습니다. DB2는 특정 함수 내용을 사용하는 UDF가 어떤 방식으로든 같은 내용을 사용하는 다른 UDF와 일치하는지를 점검하지 않습니다.

예: UDT에서의 문자열 검색

이 예는 앞 예의 연속입니다. 『예: BLOB 문자열 검색』에서 FINDSTRING 함수에 만족하지만 이제 소스 유형이 BLOB인 BOAT 구별 유형을 정의했습니다. 또한 사용자는 FINDSTRING이 BOAT 데이터 유형이 있는 값에 대한 연산을 하기를 원하므로, 다른 FINDSTRING 함수를 작성합니다. 이 함수는 『예: BLOB 문자열 검색』에 있는 BLOB 값에 대한 연산을 수행하는 FINDSTRING이 소스가 됩니다. 이 예에서 FINDSTRING의 오버로드에 유의하십시오.

```
CREATE FUNCTION FINDSTRING (BOAT, VARCHAR(200))
  RETURNS INT
  SPECIFIC "slick_fboat_mar95"
  SOURCE SPECIFIC "willie_fblob_feb95"
```

이 FINDSTRING 함수에 419 페이지의 『예: BLOB 문자열 검색』에 있는 FINDSTRING 함수로부터 다른 신호가 있으므로 이름을 오버로드하는 데 문제가 없습니다. 나중에 DDL에서의 참조를 위해 고유의 구체적인 이름을 제공하려 합니다. SOURCE절을 사용하고 있으므로, 함수 속성을 지정하는 관련 키워드 또는 EXTERNAL NAME절을 사용할 수 없습니다. 이 속성은 소스 함수로부터 가져옵니다. 마지막으로, 소스 함수를 식별할 때 419 페이지의 『예: BLOB 문자열 검색』에 명시적으로 제공된 세부 함수 이름을 사용하고 있는지 보십시오. 이것이 규정화되지 않은 참조이므로, 이 소스 함수가 상주하는 스키마는 함수 경로에 있어야 합니다. 그렇지 않으면 참조가 분석되지 않습니다.

예: UDT 매개변수가 있는 외부 함수

BOAT를 가져가 설계 속성을 검토하고 그 보트 비용을 캐나다화로 생성하기 위해 다른 UDF를 작성했습니다. 내부적으로 노동 비용이 유로화나 일본 엔 또는 미화 달러로 가격이 책정될 수 있어도 이 함수는 필요한 통화, 즉 캐나다 달러로 보트를 만드는 비용을 생성해야 합니다. 이는 환율 웹 페이지로부터 현재 환율 정보를 얻어야 하고, 대답은 웹 페이지 내용에 따라 달라짐을 의미합니다. 이는 함수를 NOT DETERMINISTIC(또는 VARIANT)로 만듭니다.

```
CREATE FUNCTION BOAT_COST (BOAT)
  RETURNS INTEGER
  EXTERNAL NAME '/u/marine/funcdir/costs!boatcost'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED
```

CAST FROM 및 SPECIFIC이 지정되지 않지만 NOT DETERMINISTIC가 지정되어야 합니다. 보안상의 여기에 다시 FENCED가 선택됩니다.

예: UDT의 AVG

이 예는 CANADIAN_DOLLAR 구별 유형에 대해 AVG 컬럼 함수를 구현합니다. CANADIAN_DOLLAR 정의에 대해서는 305 페이지의 『예: 비용』의 내용을 참조하십시오. 명백한 유형 지정은 구별 유형에 대한 내장 AVG 함수를 사용하지 못하도록 합니다. 그 결과 CANADIAN_DOLLAR에 대한 소스 유형이 DECIMAL이었음이 나타나고, 따라서 AVG(DECIMAL) 내장 함수를 소스로 하여 AVG를 구현합니다. 이 구현 기능은 DECIMAL에서 CANADIAN_DOLLAR로, CANADIAN_DOLLAR에서 DECIMAL로 유형변환할 수 있는지에 따라 달라지지만, DECIMAL이 CANADIAN_DOLLAR의 소스 유형이므로 이러한 유형변환이 기능할 것임을 압니다.

```
CREATE FUNCTION AVG (CANADIAN_DOLLAR)
  RETURNS CANADIAN_DOLLAR
  SOURCE "SYSIBM".AVG(DECIMAL(9,2))
```

SOURCE절에서 함수 이름을 규정화했고, 함수 경로에 잠재된 기타 AVG 함수가 있을 수 있습니다.

예: 카운팅

사용자의 간단한 카운팅 함수는 처음에 1을 리턴하고, 호출될 때마다 결과가 1씩 증가됩니다. 이 함수는 SQL 인수를 취하지 않고, 그 응답이 호출마다 다르므로 정의에 의해 NOT DETERMINISTIC 함수가 됩니다. 스크래치 패드를 사용하여 리턴되는 최종 값을 저장하고, 호출될 때마다 이 값을 증가시켜 리턴합니다. 이 함수를 엄격하게 테스트했고 데이터베이스에 대한 DBADM 권한을 소유하고 있으므로 이를 NOT FENCED로서 정의합니다. (DBADM은 CREATE_NOT_FENCED를 내포합니다.)

```
CREATE FUNCTION COUNTER ()
  RETURNS INT
  EXTERNAL NAME '/u/roberto/myfuncs/util!ctr'
LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC
  NOT FENCED
  SCRATCHPAD
  DISALLOW PARALLEL
```

매개변수 정의가 제공되지 않고 빈 괄호만 제공됩니다. 위 함수는 SCRATCHPAD를 지정하고, 기본 스펙인 NO FINAL CALL을 사용합니다. 이 경우, (100 바이트) 스크래치 패드 기본 크기가 충분하므로 최종 호출을 이용하여 저장영역을 비울 필요가 없으며 따라서 NO FINAL CALL이 지정됩니다. COUNTER 함수가 제대로 작동되기 위해서는 단일 스크래치 패드를 사용해야 하므로, DISALLOW PARALLEL이 추가되어 DB2가 병렬 작동되지 못하도록 합니다. 이 COUNTER 함수의 구현은 515 페이지의『예: 계수기(Counter)』의 내용을 참조하십시오.

예: OLE 자동화 오브젝트에서의 카운팅

이 예는 인스턴스 변수, nbrOfInvoke를 가진 OLE(Object Linking and Embedding) 자동화 오브젝트인 counter로서 이전의 카운팅 예를 구현하여 호출 횟수를 추적합니다. UDF가 호출될 때마다 오브젝트의 increment 메소드는 nbrOfInvoke 인스턴스 변수를 증가시키고 현재 상태를 리턴합니다. OLE 프로그램 식별자(progID) bert.bcounter로 자동화 오브젝트가 Windows 레지스트리에서 등록된 상태입니다.

```
CREATE FUNCTION bcounter ()
  RETURNS integer
  EXTERNAL NAME 'bert.bcounter!increment'
  LANGUAGE OLE
  PARAMETER STYLE DB2SQL
  SCRATCHPAD
  NOT DETERMINISTIC
  FENCED
  NULL CALL
  NO SQL
  NO EXTERNAL ACTION
  DISALLOW PARALLEL;
```

counter 클래스의 구현은 528 페이지의『예: BASIC으로 작성한 계수기(Counter) OLE 자동 UDF』 및 530 페이지의『예: C++로 작성한 계수기(Counter) OLE 자동 UDF』에 나와 있습니다. DB2에서의 OLE 지원에 대한 세부사항은 471 페이지의『OLE 자동 UDF 작성』의 내용을 참조하십시오.

예: 문서 ID를 리턴하는 테이블 함수

제공된 주제 영역(첫번째 매개변수)과 일치하며 제공된 문자열(두 번째 매개변수)이 포함된 텍스트 관리 시스템에 있는 알려진 각 문서에 대해 단일 문서 식별자

컬럼으로 구성된 행을 리턴하는 테이블 함수를 작성했습니다. 이 UDF는 텍스트 관리 시스템 함수를 사용하여 문서를 신속하게 식별합니다.

```
     CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
       RETURNS TABLE (DOC_ID CHAR(16))
       EXTERNAL NAME '/common/docfuncs/rajiv/udfmatch'
LANGUAGE C
       PARAMETER STYLE DB2SQL
       NO SQL
       DETERMINISTIC
       NO EXTERNAL ACTION
       NOT FENCED
       SCRATCHPAD
       NO FINAL CALL
       DISALLOW PARALLEL
       CARDINALITY 20
```

단일 세션의 문맥 내에서 항상 같은 테이블을 리턴하므로, DETERMINISTIC로서 정의됩니다. DOC_ID 컬럼 이름을 포함하여, DOCMATCH로부터 출력을 정의하는 RETURNS절을 보십시오. FINAL CALL은 각 테이블 함수에 대해 지정할 필요가 없습니다. 또한, 테이블 함수로서 추가되는 DISALLOW PARALLEL 키워드는 병렬로 처리되지 않습니다. DOCMATCH로부터의 출력 *크기*가 매우 가변적이어도 CARDINALITY 20이 대표값이고, DB2 최적화 알고리즘이 결정을 잘 할 수 있도록 지정됩니다.

일반적으로 이 테이블 함수는 다음과 같은 문서 텍스트가 포함된 테이블과의 조인에 사용됩니다.

```
  SELECT T.AUTHOR, T.DOCTEXT
  FROM DOCS as T, TABLE(DOCMATCH('MATHEMATICS', 'ZORN''S LEMMA')) as F
  WHERE T.DOCID = F.DOC_ID
```

FROM절에 테이블 함수를 지정하기 위한 특수 구문(TABLE 키워드)에 유의하십시오. 이 호출에서 `docmatch()` 테이블 함수는 `Zorn's Lemma`를 참조하는 각 `mathematics` 문서에 대해 단일 컬럼 DOC_ID가 포함된 행을 리턴합니다. 이 DOC_ID 값들은 저자의 이름 및 문서 텍스트를 검색하여 마스터 문서 테이블에 조인됩니다.

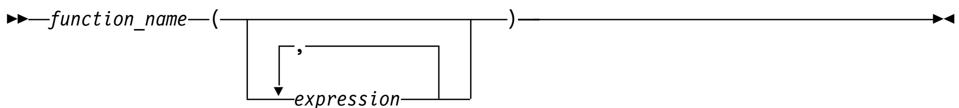
메소드 및 함수 사용

스칼라 및 컬럼 UDF 및 메소드는 표현식이 유효한 SQL문 내에서 호출될 수 있습니다.(유효성을제한하는 모든 컬럼 함수에 대한 추가 규칙이 있습니다.) 테이블 UDF는 SELECT의 FROM절에서만 참조될 수 있습니다. *SQL 참조서*에서는 이러한 모든 문맥에 대해 자세히 설명되어 있습니다. 이 절에서 사용되는 논의 및 예는 상대적으로 간단한 SELECT문 문맥을 중심으로 하지만, 그 사용은 이러한 문맥으로 제한되지 않습니다.

함수 경로 및 *함수 선택* 알고리즘의 사용 및 중요성에 대한 요약은 413 페이지의 『UDF 및 Method 개념』의 내용을 참조하십시오. *SQL 참조서*에 이 두 개념 모두에 대한 세부사항이 있습니다. 함수에 대한 DML(Data Manipulation Language) 참조의 해상도는 함수 선택 알고리즘을 사용하므로, 작동 방법을 이해하는 것이 중요합니다.

함수 참조

UDF 함수인지 내장 함수인지에 관계없이, 함수에 대한 각 참조에는 다음과 같은 구문이 들어 있습니다.



위 예에서 `function_name`은 규정화되지 않은 함수 이름 또는 규정화된 함수 이름이 될 수 있고, 인수의 수는 0에서 90까지 가능하며, 표현식에는 다음 사항이 포함될 수 있습니다.

- 컬럼 이름, 규정 또는 비규정
- 상수
- 호스트 변수
- 특수 레지스터
- 매개변수 표시문자.(매개변수 표시문자 사용시 제한사항에 대해서는, 매개변수 표시문자 규칙을 설명하는 *SQL 참조서*에 있는 절을 참조하십시오.)

인수의 위치가 중요하고, 의미가 정정되려면 함수 정의를 따라야 합니다. 인수의 위치와 함수 정의는 함수 내용 자체를 따라야 합니다. DB2는 함수 정의와 더 잘 일치하는 인수를 섞으려 하지 않으며, DB2는 개별 함수 매개변수의 의미를 이해하지 못합니다.

UDF 인수 표현식에서 컬럼 이름을 사용할 때는 컬럼이 포함된 테이블 참조에 적절한 범위가 있어야 합니다. 조인에서 참조되는 테이블 함수의 경우, 이는 다른 테이블 또는 테이블 함수로부터의 컬럼을 포함시키는 인수의 경우 그 테이블 또는 테이블 함수는 FROM절에서 참조가 포함된 테이블 함수 앞에 나와야 함을 의미합니다. 테이블 함수 인수에서의 컬럼 사용 규칙에 대한 전체 논의는 *SQL 참조서*의 내용을 참조하십시오.

함수 호출 예

함수 호출의 유효한 몇가지 예는 다음과 같습니다.

```
AVG(FLOAT_COLUMN)
BLOOP(COLUMN1)
BLOOP(FLOAT_COLUMN + CAST(? AS INTEGER))
BLOOP(:hostvar :indicvar)
BRIAN.PARSE(CHAR_COLUMN CONCAT USER, 1, 0, 0, 1)
CTR()
FLOOR(FLOAT_COLUMN)
PABLO.BLOOP(A+B)
PABLO.BLOOP(:hostvar)
"search_schema"(CURRENT FUNCTION PATH, 'GENE')
SUBSTR(COLUMN2,8,3)
SYSFUN.FLOOR(AVG(EMP.SALARY))
SYSFUN.AVG(SYSFUN.FLOOR(EMP.SALARY))
SYSIBM.SUBSTR(COLUMN2,11,LENGTH(COLUMN3))
SQRT(SELECT SUM(length*length)
     FROM triangles
     WHERE id= 'J522'
     AND legtype <> 'HYP')
```

위 함수 중에서 테이블 함수가 있는 경우 이를 참조하는 구문은 위에 제시된 구문과는 약간 차이가 있습니다. 예를 들어, PABLO.BLOOP가 테이블 함수인 경우 제대로 참조하려면 다음을 사용하십시오.

```
TABLE(PABLO.BLOOP(A+B)) AS Q
```

함수의 매개변수 표시문자 사용

주요 제한사항에는 매개변수 표시문자가 포함됩니다. 다음을 코드화할 수 없습니다.

```
BLOOP(?)
```

함수 선택 논리가 인수의 데이터 유형이 어떻게 될 지 알지 못하므로 참조를 분석할 수 없습니다. CAST 스펙을 사용하여 INTEGER와 같은 매개변수 표시문자 유형을 표시할 수 있고, 그러면 함수 선택 논리가 진행될 수 있습니다.

```
BLOOP(CAST(? AS INTEGER))
```

규정화된 함수 참조 사용

규정화된 함수 참조를 사용할 경우 일치 함수에 대한 DB2 검색을 그 스키마로 제한합니다. 예를 들어, 다음과 같은 명령문이 있습니다.

```
SELECT PABLO.BLOOP(COLUMN1) FROM T
```

PABLO 스키마에 있는 BLOOP 함수만 고려됩니다. SERGE 사용자가 BLOOP 함수를 정의했는지, 내장 BLOOP 함수가 있는지는 문제가 되지 않습니다. PABLO 사용자가 스키마에 두 개의 BLOOP 함수를 정의했다고 가정하십시오.

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS ...
CREATE FUNCTION BLOOP (DOUBLE) RETURNS ...
```

따라서 BLOOP는 PABLO 스키마 내에서 오브로드되고, 함수 선택 알고리즘은 인수의 데이터 유형, column1에 따라 최고의 BLOOP를 선택합니다. 이런 경우, 두 PABLO.BLOOP 모두 수치 인수를 취하며, column1이 수치 유형 중 하나가 아닌 경우 명령문이 실패합니다. 반대로, column1이 SMALLINT이거나 INTEGER이면 함수 선택은 첫번째 BLOOP로 되고, column1이 DECIMAL, DOUBLE, REAL 또는 BIGINT이면, 두 번째 BLOOP가 선택됩니다.

이 예에서의 몇가지 유의사항:

1. 인수의 승격을 보여줍니다. 첫번째 BLOOP는 INTEGER 매개변수로 정의되었지만 여기에 SMALLINT 인수를 전달할 수 있습니다. 함수 선택 알고리즘은 내장 데이터 유형에서의 승격을 지원하고(세부사항은 *SQL 참조서* 참조) DB2는 적절한 데이터 값 변환을 수행합니다.

2. SMALLINT 또는 INTEGER 인수로 두 번째 BLOOP를 호출하려면 다음과 같이 사용자 명령문에서 명시적인 조치를 수행해야 합니다.

```
SELECT PABLO.BLOOP(DOUBLE(COLUMN1)) FROM T
```

3. 아니면, DECIMAL 또는 DOUBLE 인수와 함께 첫번째 BLOOP를 호출하려면 사용자의 의도에 따라 명시적인 조치를 선택할 수 있습니다.

```
SELECT PABLO.BLOOP(INTEGER(COLUMN1)) FROM T
SELECT PABLO.BLOOP(FLOOR(COLUMN1)) FROM T
SELECT PABLO.BLOOP(CEILING(COLUMN1)) FROM T
SELECT PABLO.BLOOP(INTEGER(ROUND(COLUMN1,0))) FROM T
```

기타 함수를 검토하려면 *SQL 참조서*의 내용을 참조하십시오. INTEGER 함수는 SYSIBM 스키마에 있는 내장 함수입니다. FLOOR, CEILING 및 ROUND 함수는 DB2와 함께 제공되는 UDF로서, 다른 많은 유용한 함수와 함께 SYSFUN 스키마에서 찾을 수 있습니다.

규정화되지 않은 함수 참조 사용

규정화된 함수 참조 대신 규정화되지 않은 함수 참조를 사용할 경우 일반적으로 DB2는 일치하는 함수를 검색할 때 이 함수 경로를 사용하여 참조를 규정화합니다. DROP FUNCTION 또는 COMMENT ON FUNCTION 함수의 경우, 규정화되지 않았으면 현재 권한 부여 ID를 사용하여 참조가 규정화됩니다. 따라서, *함수 경로가 무엇인지, 그리고 현재 함수 경로 스키마에 어떤 충돌 함수가 존재하는지 알아야 합니다.* 예를 들어, 사용자가 PABLO이고 정적 SQL문은 다음과 같다고 가정하십시오. 여기서 COLUMN1은 INTEGER 데이터 유형입니다.

```
SELECT BLOOP(COLUMN1) FROM T
```

426 페이지의『규정화된 함수 참조 사용』에 인용된 두 개의 BLOOP 함수를 작성했고, 이 중 하나를 선택하려고 합니다. 다음과 같은 기본 함수 경로가 사용될 경우 (컬럼1이 INTEGER이므로) 첫번째 BLOOP가 선택되고, SYSIBM 또는 SYSFUN에 충돌하는 BLOOP가 없을 경우,

```
"SYSIBM","SYSFUN","PABLO"
```

그러나, 다른 목적으로 이전에 작성한 사전 처리 컴파일 및 바인딩용 스크립트를 사용하고 있다는 사실을 잊었다고 가정하십시오. 이 스크립트에서 현재 작업에 적용되지 않는 다른 이유로 다음과 같은 함수 경로를 지정하기 위해 FUNCPATH 매개변수를 명시적으로 코드화했습니다.

```
"KATHY","SYSIBM","SYSFUN","PABLO"
```

Kathy가 개인용으로 BLOOP 함수를 작성한 경우, 이 함수 선택은 Kathy의 함수로 될 가능성이 매우 높으며 사용자의 명령문은 오류 없이 실행될 것입니다. DB2는 사용자가 무엇을 수행하고 있는지 알고 있다고 전제하므로 사용자에게는 이것이 통보되지 않습니다. 사용자가 사용자 명령문으로부터 잘못된 출력을 식별하여 필요한 정정 조치를 취해야 합니다.

함수 참조 요약

규정화된 함수 참조 및 규정화되지 않은 함수 참조 모두에 대해 함수 선택 알고리즘은 다음 항목들이 있는 적용가능한 함수들, 내장 함수 및 사용자 정의 함수를 포함한 모든 함수들을 검토합니다.

- 제공된 이름
- 함수 참조의 인수와 같은 수의 정의된 매개변수
- 해당 인수 유형으로부터 승격 가능한 또는 동일한 각 매개변수

(*적용가능 함수*란 규정화된 참조의 경우 *명명된 스키마에 있는 함수*를, 규정화되지 않은 참조의 경우 *함수 경로의 스키마에 있는 함수*를 의미합니다.) 알고리즘은 우선 완전하게 일치하는 함수를 찾고, 그런 함수가 없으면 가장 일치하는 것을 찾습니다. 규정화되지 않은 참조에 한해, 동일한 두 일치 항목이 다른 스키마에 있는 경우 결정 요인으로서 현재 함수 경로가 사용됩니다. 알고리즘에 대한 세부사항은 *SQL 참조서*의 내용을 참조하십시오.

426 페이지의『규정화된 함수 참조 사용』 끝부분에 나오는 예에서 제시된 흥미로운 기능은 *함수 참조가 중첩될 수 있으며*, 동일한 함수를 참조하기도 합니다. 이는 UDF 뿐만 아니라 내장 함수의 경우에도 일반적으로 적용됩니다. 그러나, 컬럼 함수가 포함될 때 몇가지 제한사항이 있습니다.

앞의 예를 정리해 보면 다음과 같습니다.

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS INTEGER ...
CREATE FUNCTION BLOOP (DOUBLE) RETURNS INTEGER ...
```

이제 다음 DML문을 살펴 보십시오.

```
SELECT BLOOP( BLOOP(COLUMN1)) FROM T
```

column1이 DECIMAL 컬럼이거나 DOUBLE 컬럼인 경우 내부 BLOOP 참조는 위에서 정의된 두 번째 BLOOP으로 됩니다. 이 BLOOP가 INTEGER를 리턴하므로, 외부 BLOOP는 첫번째 BLOOP로 됩니다.

column1이 SMALLINT 컬럼이거나 INTEGER 컬럼인 경우 내부 bloop 참조는 위에서 정의된 첫번째 BLOOP가 됩니다. 이 BLOOP가 INTEGER를 리턴하므로, 외부 BLOOP도 첫번째 BLOOP로 됩니다. 이런 경우 동일한 함수에 대해 중첩된 참조가 나타납니다.

함수 참조에 중요한 몇가지 추가 지점은 다음과 같습니다.

- SQL 연산자 중 하나의 이름으로 함수를 정의하여 실제로 *infix 표기*를 사용한 UDF를 호출할 수 있습니다. 예를 들어, 구별 유형 BOAT를 가진 값에 대해 어떤 의미를 "+" 연산자에 연결할 수 있다고 가정하십시오. 다음과 같은 UDF를 정의할 수 있습니다.

  ```
  CREATE FUNCTION "+" (BOAT, BOAT) RETURNS ...
  ```

  그런 후 다음과 같은 유효한 SQL문을 작성할 수 있습니다.

  ```
  SELECT BOAT_COL1 + BOAT_COL2
  FROM BIG_BOATS
  WHERE BOAT_OWNER = 'Nelson Mattos'
  ```

  그러나 동등하게 유효한 명령문을 작성할 수도 있습니다.

  ```
  SELECT "+"(BOAT_COL1, BOAT_COL2)
  FROM BIG_BOATS
  WHERE BOAT_OWNER = 'Nelson Mattos'
  ```

  사용자는 >, =, LIKE, IN 등의 내장 조건 연산자를 이런 방식으로 오버로드할 수 없습니다. 나누기(/) 연산자를 오버로드하는 UDF의 예는 506 페이지의 『예 : 정수 나누기 연산자』의 내용을 참조하십시오.

- 함수 선택 알고리즘은 특정 함수로 되는 데 있어서 참조의 문맥을 고려하지 않습니다. 이전과 약간 다르게 수정된 이 BLOOP 함수를 보십시오.

  ```
  CREATE FUNCTION BLOOP (INTEGER) RETURNS INTEGER ...
  CREATE FUNCTION BLOOP (DOUBLE) RETURNS CHAR(10)...
  ```

이제 다음과 같은 SELECT문을 작성합니다.

```
SELECT 'ABCDEFG' CONCAT BLOOP(SMALLINT_COL) FROM T
```

SMALLINT 인수를 사용하여 검색된 가장 일치하는 함수가 위에 정의된 첫번째 BLOOP이므로, CONCAT의 두 번째 피연산자는 INTEGER 데이터 유형으로 됩니다. CONCAT는 문자열 인수를 요구하므로 명령문이 실패합니다. 첫 번째 BLOOP가 없는 경우에는 기타 BLOOP가 선택되고 명령문 실행이 성공합니다.

명령문을 실패시키는 또다른 유형의 문맥 불일치는, 제공된 함수 참조가 스칼라 또는 컬럼 함수가 필요한 문맥에서 테이블 함수로 되는 경우입니다. 그 반대의 경우도 발생할 수 있습니다. 참조는 테이블 함수가 필요할 때 스칼라 또는 컬럼 함수로 됩니다.

- LOB 유형: BLOB, CLOB 또는 DBCLOB를 가진 결과 또는 매개변수로 UDF 및 메소드를 정의할 수 있습니다. 값의 소스가 *LOB 위치 지정자* 호스트 변수인 경우에도 그러한 함수를 호출하기 전에 DB2는 저장영역의 전체 LOB 값을 구체화합니다. 예를 들어, C언어 응용프로그램의 다음 부분을 살펴 보십시오.

  ```
  EXEC SQL BEGIN DECLARE SECTION;
    SQL TYPE IS CLOB(150K) clob150K ;        /* LOB host var */
    SQL TYPE IS CLOB_LOCATOR clob_locator1;  /* LOB locator host var */
    char                   string[40];       /* string host var */
  EXEC SQL END DECLARE SECTION;
  ```

  호스트 변수 :clob150K 또는 :clob_locator1이 해당 매개변수가 CLOB(500K)로서 정의된 함수에 대한 인수로서 유효합니다. 따라서, 418 페이지의 『예: 문자열 검색』에 정의된 FINDSTRING을 참조하므로 프로그램에서 다음 두 항목이 모두 유효합니다.

  ```
  ... SELECT FINDSTRING (:clob150K, :string) FROM ...
  ... SELECT FINDSTRING (:clob_locator1, :string) FROM ...
  ```

- LOB 유형 중 하나를 가진 UDF 매개변수 또는 결과는 AS LOCATOR 수정자를 이용하여 작성할 수 있습니다. 이런 경우 전체 LOB 값은 호출 이전에 구체화되지 않습니다. 대신, LOB LOCATOR가 UDF로 전달되고, UDF는 특수 UDF API를 사용하여 LOB 값의 실제 바이트 수를 조작할 수 있습니다.(세부 사항은 493 페이지의 『UDF 매개변수 또는 결과로 LOB 위치 지정자 사용』의 내용을 참조하십시오.)

또한 LOB에 기초한 구별 유형을 가진 결과 또는 UDF 매개변수에 대해 이 기능을 사용할 수도 있습니다. 이 기능은 비분리로서 정의된 UDF에 제한됩니다. 그러한 함수에 대한 인수는 정의된 유형의 LOB 값이 될 수 있습니다. 즉, LOCATOR 유형 중 하나로서 정의된 호스트 변수가 아니어도 됩니다. 인수로서 호스트 변수 위치 지정자를 사용하는 것은 UDF 매개변수의 AS LOCATOR 및 결과 정의를 사용하는 것과 완전히 대비됩니다.

- UDF 및 메소드는 매개변수 또는 결과로서 구별 유형으로 정의될 수 있습니다.(이전 예에서 보여줬습니다.) DB2는 구별 유형의 소스 데이터 유형 형식으로 값을 UDF로 전달합니다.

  호스트 변수에서 시작되며 해당 매개변수가 구별 유형으로서 정의된 UDF로 인수로서 사용되는 구별 유형 값은 **사용자에 의해 구별 유형으로 명시적으로 유형변환되어야 합니다**. 구별 유형에 대한 호스트 언어 유형은 없습니다. DB2의 명백한 유형 지정의 경우 이것이 필수적입니다. 그렇지 않을 경우 사용자 결과가 명확할 수 있습니다. 따라서, BLOB를 통해 정의되어 있는 BOAT 구별 유형을 고려하고, 인수로서 BOAT 유형의 오브젝트를 취하는 420 페이지의 『예: UDT 매개변수가 있는 외부 함수』에서 BOAT_COST UDF를 고려하십시오. C 언어 응용프로그램의 다음 부분에서, :ship 호스트 변수는 BOAT_COST 함수로 전달된 BLOB 값을 보유합니다.

```
        EXEC SQL BEGIN DECLARE SECTION;
      SQL TYPE IS BLOB(150K) ship;
 EXEC SQL END DECLARE SECTION;
```

  다음 명령문 모두가 :ship 호스트 변수를 BOAT 유형으로 변환하므로 BOAT_COST 함수로 됩니다.

```
... SELECT BOAT_COST (BOAT(:ship)) FROM ...
... SELECT BOAT_COST (CAST(:ship AS BOAT)) FROM ...
```

  데이터베이스에 여러 개의 BOAT 구별 유형이 있거나 다른 스키마에 BOAT UDF가 있는 경우, 함수 경로에 주의해야 합니다. 그렇지 않을 경우 사용자 결과가 명확할 수 있습니다.

제15장 사용자 정의 함수(UDF)

설명 433
DB2 및 UDF간의 인터페이스 435
인수를 DB2에서 UDF로 전달 435
UDF 인수 사용 요약 451
SQL 데이터 유형이 UDF로 전달되는 방법 453
32-비트 및 64-비트 플랫폼에서의 스크래치 패드 작성 464
UDF include 파일: sqludf.h 465
Java 사용자 정의 함수 작성 및 사용 . . . 466
Java UDF 코딩 466
Java UDF 실행 방법 변경 468
Java에 대한 테이블 함수 실행 모델 . . 469
OLE 자동 UDF 작성 471
OLE 자동화 UDF 작성 및 등록 472
오브젝트 인스턴스 및 스크래치 패드 고려사항 473
SQL 데이터 유형이 OLE 자동화 UDF로 전달되는 방식 474
BASIC 및 C++에서 OLE 자동화 구현 475
BASIC의 OLE 자동 UDF 476
C++의 OLE 자동 UDF 476
OLE DB 테이블 함수 479
OLE DB 테이블 함수 작성 480
완전한 행 집합 이름. 482
OLE DB 제공자의 서버 이름 정의. . . 484
사용자 맵핑 정의. 484
지원되는 OLE DB 데이터 유형 485
스크래치 패드 고려사항. 488
테이블 함수 고려사항 490
테이블 함수 오류 처리 491
스칼라 함수 오류 처리 492
UDF 매개변수 또는 결과로 LOB 위치 지정자 사용 493
LOB 위치 지정자 사용에 대한 시나리오 498
기타 코딩시 고려사항 499
도움말 및 추가 정보. 499
UDF 제한사항 및 조건. 502
UDF 코드 예 505
예: 정수 나누기 연산자. 506
예: CLOB 접기, 모음 찾기 510
예: 계수기(Counter) 515
예: 날씨 테이블 함수 517
예: LOB 위치 지정자를 사용하는 함수 525
예: BASIC으로 작성한 계수기(Counter) OLE 자동 UDF 528
예: C++로 작성한 계수기(Counter) OLE 자동 UDF 530
예: BASIC으로 작성한 메일 OLE 자동 테이블 함수 532
UDF 디버깅 535

설명

이 절에서는 UDF 작성 및 메소드에 대해 설명합니다. UDF의 코딩 규칙과 메소드에는 다음과 같은 차이점이 있지만 같습니다.

- DB2는 각 메소드를 지정된 구조화된 유형을 연관시키며, DB2에서 메소드로 전달된 첫번째 인수는 항상 메소드를 호출한 구조화된 유형의 인스턴스입니다.

© Copyright IBM Corp. 1993, 2000

- UDF와는 달리 메소드는 테이블을 리턴할 수 없습니다. 메소드를 FROM절에 대한 인수로 호출할 수 없습니다.

UDF 및 메소드 작성에 대한 지침은 앞에서 설명된 차이를 제외하고는 같으므로 UDF 및 메소드 작성에 대한 나머지 설명은 UDF와 메소드를 참조합니다.

간단한 표현식만 들어 있는 UDF와 같은 작은 UDF에 대해 *SQL 본문 UDF*를 사용하십시오. SQL 본문 UDF를 작성하려면, 외부 UDF를 포인트하는 대신 SQL을 사용하여 작성된 메소드 본문을 포함하는 CREATE FUNCTION 또는 CREATE METHOD문을 실행하십시오. SQL 본문 UDF를 통해 외부 언어 또는 컴파일러 없이 단일 단계에서 UDF를 선언하고 정의할 수 있습니다. SQL 본문 UDF는 DB2 최적화 알고리즘으로 액세스 가능한 SQL을 사용하여 메소드 본문이 작성되므로 성능이 향상될 수 있습니다.

다음 예에서는 SQL 본문 UDF를 작성하는 간단한 CREATE FUNCTION문을 보여줍니다.

```
CREATE FUNCTION tan(double x)
RETURNS double
NO EXTERNAL ACTION
DETERMINISTIC
LANGUAGE SQL
CONTAINS SQL
RETURN sin(x) / cos(x);
```

이 함수에 대해서는 *SQL 참조서*에서 자세한 내용을 참조하십시오.

DB2와 UDF간의 인터페이스에 대한 논의가 이루어졌다면, 이제 남은 것은 UDF를 구현하는 방법에 대한 논의입니다. UDF 작성에 관한 정보에서 기본 고려사항 중 하나로 스크래치 패드가 있는지의 여부를 강조합니다.

이 절에서의 일반적인 고려사항은 다음과 같습니다.

- UDF 정의 및 사용에 대한 중요한 자료는 409 페이지의『제14장 사용자 정의 함수(UDFs)와 메소드』에 나와 있으며, 여기에서는 반복 설명되지 않습니다. 여기에서는 UDF를 구현하는 방법에 대하여 주로 설명합니다.
- C, C++ 또는 Java에서 작성된 *외부 UDF*를 구현하려면 다음과 같은 단계를 수행해야 합니다.
  - UDF 작성.

- UDF 컴파일.
- UDF 링크.
- CREATE FUNCTION문으로 UDF 등록.
- UDF 테스트 및 디버그.

  UDF 컴파일 및 링크에 관한 정보는 *응용프로그램 빌드 안내서*의 내용을 참조하십시오.

- 471 페이지의 『OLE 자동 UDF 작성』에서 설명된 대로 OLE(Object Linking and Embedding)를 사용하여 UDF를 호출할 수 있습니다.
- CREATE FUNCTION문 만을 사용하여 OLE DB 데이터 소스로부터 테이블을 리턴하는 함수인 *OLE DB 테이블 함수*를 정의할 수 있습니다. OLE DB 테이블 함수에 대한 자세한 정보는 479 페이지의 『OLE DB 테이블 함수』의 내용을 참조하십시오.

외부 UDF와는 다른 *소스 UDF*는 별도의 코드 형태로 구현될 필요가 없음에 유의하십시오. 이러한 UDF는 다른 많은 속성들과 함께 소스 함수와 동일한 구현을 사용합니다.

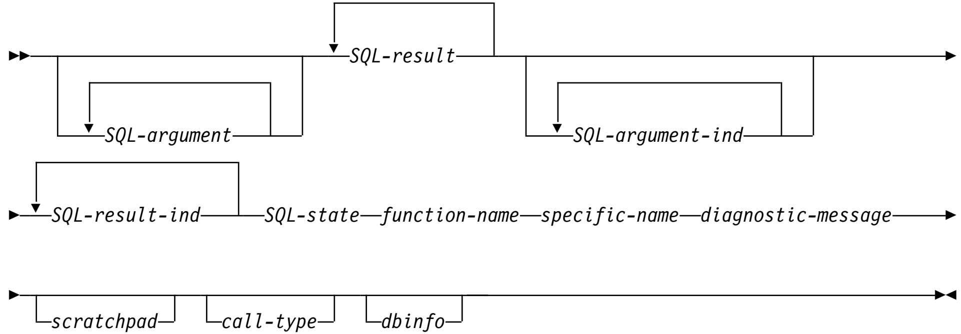
DB2 및 UDF간의 인터페이스

이 절에서는 DB2와 UDF간의 인터페이스 세부사항과 인터페이스를 관리할 수 있도록 하는 `sqludf.h` Include 파일에 대해 설명됩니다. 이 Include 파일은 C 및 C++ UDF에만 적용됩니다. Java로 UDF를 코딩하는 것에 대한 정보는 466 페이지의 『Java UDF 코딩』의 정보를 참조하십시오.

인수를 DB2에서 UDF로 전달

DB2는 함수에 대한 DML 참조에서 지정된 SQL 인수와 함께 추가 인수를 외부 UDF에 전달합니다. C 및 C++의 경우, 이들 인수들은 436 페이지의 인수를 UDF로 전달 순서대로 전달됩니다. Java UDF는 *SQL-argument*와 *SQL-result* 인수만을 취하지만, 다른 메소드를 사용하여 다른 정보에 액세스할 수 있습니다. Java UDF는 아래에 설명된 *SQL-state*와 *diagnostic-message* 인수에 대해 동일한 제한사항이 적용됩니다. Java로 UDF를 코딩하는 것에 대한 정보는 466 페이지의 『Java UDF 코딩』의 정보를 참조하십시오.

인수를 UDF로 전달하기 위한 구문



주: 외부 함수로 전달된 위의 인수들은 각각 실제 값이 아닌 해당 값에 대한 포인터입니다.

이 인수들은 다음과 같습니다.

SQL-argument

이 인수는 UDF를 호출하기 전에 DB2에 의해 설정됩니다. 이 값은 *n*번 반복되는데, 여기서 *n*은 함수 참조에 지정된 인수의 수입니다. 이들 인수 각각의 값은 함수 호출에 지정된 표현식으로부터 가져옵니다. 이들은 CREATE FUNCTION문의 해당 매개변수 정의의 데이터 유형으로 표현됩니다. 이들 데이터 유형이 C 언어 구조에 맵핑되는 방법은 453 페이지의 『SQL 데이터 유형이 UDF로 전달되는 방법』에 설명되어 있습니다.

DB2는 데이터 유형과 서버 플랫폼에 따라 *SQL-argument*에 대해 데이터를 정렬합니다.

SQL-result

이 인수는 DB2로 리턴하기 전에 UDF에 의해 설정됩니다. 스칼라 함수는 단 하나의 SQL-result를 갖습니다. 테이블 함수의 경우, CREATE FUNCTION문의 RETURNS TABLE절에 정의된 함수의 각 결과 컬럼에 대하여 하나씩의 SQL-result가 있습니다. 이들은 RETURNS TABLE 절에 정의된 컬럼에 대하여 위치에 의해 대응합니다. 즉, 첫번째 SQL-result 인수는 RETURNS TABLE절 등에 정의된 첫번째 컬럼에 대응합니다.

스칼라 함수와 테이블 함수 모두의 경우, DB2는 버퍼를 할당하고, 해당 주소를 UDF로 전달합니다. UDF는 각 결과 값을 버퍼에 넣습니다. 해당 데이터 유형으로 표현된 값을 포함하기에 충분한 버퍼 공간이 DB2에 의해 할당됩니다. 스칼라 함수의 경우, 이 데이터 유형은 CAST FROM절이 있는 경우 여기에 정의되고, CAST FROM절이 없는 경우 RETURNS 절에 정의됩니다. 테이블 함수의 경우, 데이터 유형은 RETURNS TABLE(...) 절에 정의됩니다. 이들 유형이 C 언어 구조에 맵핑되는 방법에 대한 정보는 453 페이지의 『SQL 데이터 유형이 UDF로 전달되는 방법』의 내용을 참조하십시오.

테이블 함수의 경우, DB2는 모든 계산 결과 컬럼이 DB2로 리턴될 필요가 없는 성능 최적화를 정의합니다. 이 기능을 활용하기 위해 UDF를 작성하는 경우, 테이블 함수를 참조하는 명령문에 필요한 컬럼만을 리턴합니다.

100개의 결과 컬럼을 사용하여 정의된 테이블 함수에 대한 CREATE FUNCTION문을 예로 듭니다. 이 함수를 참조하는 해당 명령문이 이 컬럼들 중 두 개에만 관심이 있다면, 이 최적화 기능을 통해 UDF는 각 열에 대하여 이들 두 개의 컬럼만을 리턴하고, 나머지 98개의 컬럼에 대하여 시간을 소비하지 않습니다. 이 최적화에 대한 자세한 정보는 아래의 *dbinfo* 인수를 참조하십시오.

리턴된 각 값(즉, 스칼라 함수의 경우 단일 값이고, 일반적으로 테이블 함수의 경우 복수 값)에 대하여, UDF 코드는 데이터 유형과 결과의 길이에 필요한 것 이상을 리턴해서는 안됩니다. DB2는 UDF 본문이 결과 버퍼 끝을 넘어서 기록되었는지 여부의 판별을 시도하고, SQLCODE -450(SQLSTATE 39501)을 리턴합니다. 그러나 DB2가 탐색하지 못한 UDF의 주요 겹쳐쓰기가 예상치 못한 결과 또는 비정상 종료를 일으킬 수 있습니다.

DB2는 데이터 유형과 서버 플랫폼에 따라 *SQL-result*에 대해 데이터를 정렬합니다.

SQL-argument-ind

이 인수는 UDF를 호출하기 전에 DB2에 의해 설정됩니다. 이것은 해당 *SQL-argument*가 널(NULL)인지 여부를 판별하기 위해 UDF에 의해 사

용될 수 있습니다. *n*번째 *SQL-argument-ind*가 *n*번째 *SQL-argument*(위에서 설명)에 대응합니다. 여기에는 다음과 같은 값이 포함됩니다.

0 이 인수가 존재하고 널(NULL)이 아닙니다.

-1 이 인수가 존재하고 그 값이 널(NULL)입니다.

함수가 NOT NULL CALL을 사용하여 정의된 경우, UDF 본문은 널(NULL) 값을 확인할 필요가 없습니다. 그러나 NULL CALL을 사용하여 정의된 경우, 어떤 인수든 NULL이 될 수 있으며, UDF는 이것을 확인해야 합니다.

표시기는 SMALLINT 값의 형태이며, 이것은 453 페이지의 『SQL 데이터 유형이 UDF로 전달되는 방법』에 설명된 대로 사용자 UDF에 정의될 수 있습니다. DB2는 데이터 유형과 서버 플랫폼에 따라 *SQL-argument-ind*에 대해 데이터를 정렬합니다.

SQL-result-ind

이 인수는 DB2로 리턴하기 전에 UDF에 의해 설정됩니다. 각 *SQL-result* 인수는 다음 중 하나가 될 수 있습니다.

이 인수는 특정 결과 값이 널(NULL)인지 여부를 알리기 위해 UDF에 의해 사용됩니다.

0 또는 양수

결과가 널(NULL)이 아닙니다.

음수 결과가 널(NULL) 값입니다. 자세한 내용은 음수 SQL-result-ind 값 해석의 내용을 참조하십시오.

음수 SQL-result-ind 값 해석:

DB2는 다음이 참인 경우, 함수 결과를 널(-2)로 취급합니다.

- 데이터베이스 구성 매개변수 DFT_SQLMATHWARN이 'YES'인 경우.
- 산술 오류로 인해 입력 인수 중 하나가 널(NULL)인 경우.
- `SQL-result-ind`가 음수인 경우.

또한 NOT NULL CALL 옵션을 사용하여 함수를 정의한 경우도 참이 됩니다.

NOT NULL CALL을 사용하여 함수가 정의된 경우에도, UDF 본문은 결과 표시기를 설정해야 합니다. 예를 들어, 나누기 함수는 분모가 0일 때, 결과를 널(NULL)로 설정할 수 있습니다.

표시기는 SMALLINT 값의 형태를 띠며, 이것은 453 페이지의『SQL 데이터 유형이 UDF로 전달되는 방법』에 설명된대로 사용자 UDF에 정의될 수 있습니다.

UDF가 RESULT 컬럼 목록을 사용하여 테이블 함수 최적화를 사용하면 필수 컬럼에 해당되는 표시기만 설정되어야 합니다.

DB2는 데이터 유형과 서버 플랫폼에 따라 *SQL-결과-ind*에 대해 데이터를 정렬합니다.

SQL-state

이 인수는 DB2로 리턴하기 전에 UDF에 의해 설정됩니다. 이것은 CHAR(5) 값의 형태입니다. UDF의 인수 정의는 453 페이지의『SQL 데이터 유형이 UDF로 전달되는 방법』에서 설명된대로 CHAR(5)에 적절하며, 경고 또는 오류 조건을 알리기 위해 UDF에 의해 사용됩니다. 이 함수가 호출될 때 여기에는 '00000' 값이 포함됩니다. UDF는 이 값을 다음에 설정할 수 있습니다.

00000 함수 코드가 경고나 오류 상태를 발견하지 못했습니다.

01H*xx* 함수 코드가 경고 상태를 발견했습니다. 이것은 SQL 경고 SQLCODE +462(SQLSTATE 01H*xx*)를 발생시킵니다. 여기에서 '*xx*'는 임의의 문자열입니다.

02000 테이블 함수에 대한 FETCH 호출에만 유효하며, 테이블에 행이 없음을 의미합니다.

38502 UDF 본문이 SQL 호출을 실행하고 오류 SQLCODE -487(SQL이 UDF에서 허용되지 않으므로 SQLSTATE 38502)을 수신하고, 이 오류를 DB2로 전달하기로 선택한 경우의 특별한 값.

기타 38*xxx*

함수 코드가 오류 상태를 발견했습니다. 이것은 SQL 오류, SQLCODE -443(SQLSTATE 38*xxx*)을 발생시킵니다. 여기에서 '*xxx*'는 임의의 문자열입니다. 380*xx*에서 384*xx*까지의 값들은

SQL92 국제 표준에 대한 드래프트 확장에 의해 예약되어 있는 값이므로 사용할 수 없으며, 385*xx*는 IBM에 의해 예약되어 있으므로 사용할 수 없습니다.

기타 다른 값은 오류 상태로 처리되어 SQLCODE -463(SQLSTATE 39001)을 발생시킵니다.

function-name

이 인수는 UDF를 호출하기 전에 DB2에 의해 설정됩니다. 이것은 DB2에서 UDF 코드로 전달되는 완전한 함수 이름입니다. 이 변수는 VARCHAR(27) 값의 형태입니다. UDF의 인수 정의는 VARCHAR(27)에 적절해야 합니다. 자세한 내용은 453 페이지의 『SQL 데이터 유형이 UDF로 전달되는 방법』의 내용을 참조하십시오.

전달되는 함수 이름의 형식은 다음과 같습니다.

```
<schema-name>.<function-name>
```

각 부분은 마침표로 구분됩니다. 다음과 같은 두 가지 예가 있습니다.

```
PABLO.BLOOP        WILLIE.FINDSTRING
```

이 형식을 사용하여 사용자는 동일한 UDF 본문을 복수의 외부 함수에 사용할 수 있으며, 호출될 때 이 함수들을 구별할 수 있습니다.

주: 오브젝트 이름과 스키마 이름에 마침표를 포함시킬 수 있으나, 권장되지는 않습니다. 예를 들어, 함수 `rotate`가 스키마 `obj.op`에 있는 경우 리턴되는 함수 이름은 `obj.op.rotate`인데, 이러한 경우, 스키마 이름이 `obj`인지 또는 `obj.op`인지 분명하지 않습니다.

specific-name

이 인수는 UDF를 호출하기 전에 DB2에 의해 설정됩니다. 이것은 DB2에서 UDF 코드로 전달되는 함수의 고유한 이름입니다. 이 변수는 VARCHAR(18) 값의 형태입니다. UDF의 인수 정의는 VARCHAR(18)에 적절해야 합니다. 자세한 내용은 453 페이지의 『SQL 데이터 유형이 UDF로 전달되는 방법』의 내용을 참조하십시오. 다음과 같은 두 가지 예가 있습니다.

```
willie_find_feb99      SQL9904281052440430
```

첫번째 값은 사용자의 CREATE FUNCTION문에서 사용자가 지정합니다. 사용자가 값을 지정하지 않으면 둘째 값은 현재 시간소인에서 DB2에 의해 생성됩니다.

function-name 인수에서와 같이, 이 값을 전달하는 이유는 UDF에게 어떤 세부 함수가 이것을 호출했는지를 정확히 식별할 수 있는 수단을 제공하기 위한 것입니다.

diagnostic-message

이 인수는 DB2로 리턴하기 전에 UDF에 의해 설정됩니다. UDF는 이 인수를 사용하여 DB2 메시지에 메시지 텍스트를 삽입할 수 있습니다. 이것은 VARCHAR(70) 값의 형태입니다. UDF의 인수 정의는 VARCHAR(70)에 적절해야 합니다. 자세한 내용은 453 페이지의 『SQL 데이터 유형이 UDF로 전달되는 방법』의 내용을 참조하십시오.

UDF가 위에서 설명한 *SQL-state* 인수를 사용하여 오류나 경고를 리턴할 때, 여기에는 설명 정보가 포함될 수 있습니다. DB2는 이 정보를 메시지의 토큰으로 포함합니다.

DB2는 UDF를 호출하기 전에 첫번째 문자를 널(NULL)로 설정합니다. 리턴시, 이 문자열은 C의 널(NULL)로 종료하는 문자열로 처리됩니다. 이 문자열은 오류 조건의 토큰으로 SQLCA에 포함됩니다. 최소한 이 문자열의 첫번째 부분이 SQLCA나 DB2 CLP 메시지에 표시됩니다. 그러나 표시되는 실제 문자 수는 다른 토큰의 길이에 따라 달라지는데, 이것은 DB2가 SQLCA가 지정하는 전체 토큰 길이에 대한 한계를 준수하기 위해 토큰을 절단할 수 있기 때문입니다. X'FF'는 SQLCA에서 토큰을 구분하는 데 사용되므로, 텍스트에서는 이 문자를 사용하지 마십시오.

UDF 코드는 전달된 VARCHAR(70) 버퍼의 용량보다 더 많은 텍스트를 리턴해서는 안됩니다. DB2는 UDF 본문이 이 버퍼 끝 너머에 기록되었는지 여부의 판별을 시도하고, SQLCODE -450(SQLSTATE 39501)을 리턴합니다. 그러나 UDF에 의한 겹쳐쓰기는 DB2에 의해 발견되지 않을 수도 있으므로, 예상치 못한 결과나 이상 종료를 유발할 수 있습니다.

DB2는 UDF에서 DB2로 리턴되는 메시지 토큰은 데이터베이스와 같은 코드 페이지에 있는 것으로 간주합니다. 사용자의 UDF는 이 사실을 확인해야 합니다. 사용자가 7 비트 불변 ASCII 부속 집합을 사용하는 경우, 사용자의 UDF는 어떤 코드 페이지로든 메시지 토큰을 리턴할 수 있습니다.

scratchpad

이 인수는 UDF를 호출하기 전에 DB2에 의해 설정됩니다. 이것은 UDF에 대한 CREATE FUNCTION문이 SCRATCHPAD 키워드를 지정한 경우에만 존재합니다. 이 인수는 LOB 데이터 유형 값을 전달하는 데 사용되는 구조와 똑같은 구조로서, 다음과 같은 요소들이 포함됩니다.

- 스크래치 패드의 길이를 포함하는 INTEGER. 스크래치 패드의 길이를 변경하면 SQLCODE -450(SQLSTATE 39501)이 됩니다.
- 실제 스크래치 패드는 다음과 같이 모든 2진 0으로 초기화됩니다.

  스칼라 함수의 경우, 첫번째 호출 이전에 초기화되며, 보통 그 이후에 DB2에 의해 검색되거나 수정되지 않습니다.

  테이블 함수의 경우, CREATE FUNCTION에 FINAL CALL이 지정된 경우, 위에서와 같이 UDF로의 첫번째 호출 이전에 초기화됩니다. 이 호출 이후 스크래치 패드 내용은 완전히 테이블 함수의 제어 하에 놓이게 됩니다.

  NO FINAL CALL이 지정되었거나 테이블 함수의 디폴트인 경우, 스크래치 패드는 위에서와 같이 각 OPEN 호출에 대하여 초기설정되고, 스크래치 패드 내용은 OPEN 호출들 사이에서 테이블 함수의 제어를 받게됩니다. 이것은 조인이나 부속 조회에서 사용되는 테이블 함수의 경우 매우 중요합니다. 여러개의 OPEN 호출들에서 스크래치 패드의 내용을 유지보수할 필요가 있는 경우, 사용자의 CREATE FUNCTION 문에 FINAL CALL을 지정해야 합니다. 일반적인 OPEN, FETCH 및 CLOSE 호출과 함께 FINAL CALL을 지정하면, 스크래치 패드 유지보수 및 자원 릴리스의 목적으로 테이블 함수 역시 FIRST와 FINAL 호출을 수신하게 됩니다.

전달된 인수가 같은 구조를 가지고 있으므로, CLOB나 BLOB 중 하나와 동일한 유형을 사용하여 UDF에 스크래치 패드를 맵핑할 수 있습니다. 자세한 내용은 453 페이지의 『SQL 데이터 유형이 UDF로 전달되는 방법』의 내용을 참조하십시오.

UDF 코드는 스크래치 패드 버퍼 외부에서는 변경을 수행하지 않음을 주의하십시오. DB2는 UDF 본문이 이 버퍼 끝 이후로 기록했는지를 몇 몇 문자, SQLCODE -450(SQLSTATE 39501)으로 판별하려 하는데, UDF에 의한 주요 겹쳐쓰기가 예상치 못한 결과나 이상 종료를 발생시킬 수 있으며, 발생되는 장애가 DB2에서 허용되지 않는 것일 수 있습니다.

스크래치 패드를 사용하는 스칼라 UDF가 부속 조회에서 참조된 경우, DB2는 부속 조회 호출들 사이에서 스크래치 패드를 새로 고치기로 결정할 수도 있습니다. 이 새로 고침은 UDF에 FINAL CALL이 지정된 경우, *최종 호출*이 이루어진 후에 발생합니다.

DB2는 데이터 필드가 데이터 유형의 저장에 대해 정렬되도록 스크래치 패드를 초기화합니다. 이렇게 하면 길이 필드를 포함한 전체 스크래치 패드가 제대로 정렬되지 않습니다. 스크래치 패드 선언 및 액세스에 대한 자세한 내용은 464 페이지의 『32-비트 및 64-비트 플랫폼에서의 스크래치 패드 작성』을 참조하십시오.

call-type

이 인수는, 존재하는 경우, UDF를 호출하기 전에 DB2에 의해 설정됩니다. 스칼라 함수의 경우, 이 인수는 CREATE FUNCTION문에 FINAL CALL이 지정된 경우에만 존재하지만, 테이블 함수의 경우 항상 존재합니다. 이것은 *scratchpad* 인수 뒤에 나오거나, *scratchpad* 인수가 없는 경우 *diagnostic-message* 인수 뒤에 나옵니다. 이 인수는 INTEGER 값의 형식입니다. UDF의 인수 정의는 INTEGER에 적절해야 합니다. 자세한 내용은 453 페이지의 『SQL 데이터 유형이 UDF로 전달되는 방법』의 내용을 참조하십시오.

현재 사용할 수 있는 모든 값들이 아래에 나열되어 있어도, 사용자의 UDF에 ″if A do AA, else if B do BB, else it must be C so do CC″ 유형의 논리가 아닌, 예상되는 모든 값들을 명백하게 테스트하는 switch나 case문이 포함되어야 합니다. 이것은 추가적인 호출 유형이 앞으로 추가

될 수도 있으며, 명백하게 조건 C에 대하여 테스트하지 않는 경우, 새로운 가능성이 추가될 때 문제가 발생할 수 있기 때문입니다.

주:

1. 모든 호출-유형에 대해 UDF가 *SQL-state* 및 *diagnostic-message* 리턴 값을 설정하는 것이 적절할 수 있습니다. 이 정보는 다음의 각 유형에 대한 설명에서 반복되지 않습니다. 모든 호출에 대하여 DB2는 이 인수들에 대하여 앞서 설명된 지정된 조치를 취하게 됩니다.
2. include 파일 `sqludf.h`는 UDF와 함께 사용되며, 465 페이지의『UDF include 파일: sqludf.h』에 설명되어 있습니다. 이 파일에는 상수로 해석되는 다음과 같은 call-type 값에 대한 기호 정의가 들어 있습니다.

스칼라 함수에 대해 *call-type*에는 다음이 포함됩니다.

-1
: 이것은 이 명령문에서 UDF에 대한 *FIRST* 호출입니다. 스크래치 패드가 있는 경우, 스크래치 패드는 UDF가 호출될 때 2진 0으로 설정됩니다. 모든 인수 값이 전달되고, UDF는 한 번의 초기화 조치를 필요로 하는 것이라면 어떤 것이든 해야 합니다. 또한 스칼라 UDF에 대한 FIRST 호출은 응답을 작성하여 리턴해야 한다는 점에서 NORMAL 호출과 같습니다.

  SCRATCHPAD가 지정되었으나 FINAL CALL이 지정되지 않은 경우, UDF는 최초의 호출을 식별하기 위한 이러한 call-type 인수를 갖지 않습니다. 대신 스크래치 패드의 완전 제로(all-zero) 상태에 의존해야 합니다.

0
: 이것은 *NORMAL* 호출입니다. 모든 SQL 입력 값이 전달되고 UDF는 결과를 작성 및 리턴해야 합니다. UDF는 *SQL-state* 및 *diagnostic-message* 정보를 리턴할 수도 있습니다.

1
: 이것은 *FINAL* 호출로서, *SQL-argument*나 *SQL-argument-ind* 값이 전달되지 않으며, 이들 값을 조사하려는 시도가 예상치 못한 결과를 발생시킬 수 있습니다. 스크래치 패드도 전달되는 경우, 이것은 이전 호출에서 건드리지 않습니다. UDF는 이 시점에서 자원을 릴리스하게 됩니다.

자원 릴리스

스칼라 UDF는 예를 들어 메모리와 같은 자신이 필요로 했던 자원을 릴리스하게 됩니다. FINAL CALL이 UDF에 지정되면, SCRATCHPAD도 역시 지정되고 자원을 추적하는 데 사용된 경우, 해당 FINAL 호출이 자원을 릴리스하는 위치가 됩니다. FINAL CALL이 지정되지 않은 경우, 획득한 자원은 동일한 호출에서 릴리스되어야 합니다.

테이블 함수의 경우 *call-type*에는 다음이 포함됩니다.

-2
이것은 *FIRST* 호출로서, UDF에 FINAL CALL 키워드가 지정된 경우에만 발생합니다. 스크래치 패드는 이 호출 전에 2진 0으로 설정됩니다. 인수 값이 테이블 함수로 전달되고, 메모리를 확보할 것인지 또는 다른 1회성의 자원 초기화를 수행할 것인지를 선택할 수 있습니다. 이것은 OPEN 호출이 아니며, 이것이 OPEN 호출 뒤에 나옴에 주의하십시오. 최초 호출에서 테이블 함수는 DB2가 데이터를 무시하므로 DB2에 어떠한 데이터도 리턴해서는 안됩니다.

-1
이것은 *OPEN* 호출입니다. NO FINAL CALL이 지정된 경우 스크래치 패드가 초기화되지만, 그렇지 않은 경우, 반드시 초기화되지 않는 것은 아닙니다. 모든 SQL 인수 값이 OPEN시 테이블 함수로 전달됩니다. 테이블 함수는 OPEN 호출시 DB2로 어떠한 데이터도 리턴해서는 안됩니다.

0
이것은 FETCH 호출이며, DB2에서는 테이블 함수가 리턴 값 세트로 구성되는 한 행이거나 SQLSTATE 값 '02000'으로 표시되는 테이블 끝 조건 중 어느 하나를 리턴해야 합니다. 스크래치 패드가 UDF로 전달되는 경우, 입력시 이전 호출에서 건드리지 않습니다.

1
이것은 테이블 함수에 대한 CLOSE 호출입니다. 이것은 OPEN 호출과 짝을 이루며, 외부 CLOSE 처리(예를 들어 소스 파일 닫기) 및 자원 릴리스(특히 NO FINAL CALL 케이스에 대하여)를 수행하는 데 사용될 수 있습니다.

조인이나 부속 조회를 사용하는 경우, OPEN/FETCH.../CLOSE 호출 절차는 명령문 실행 내에서 반복될 수 있으나, 단 하나의

FIRST 호출과 단 하나의 FINAL 호출만이 있습니다. FIRST와 FINAL 호출은 해당 테이블 함수에 대하여 FINAL CALL이 지정된 경우에만 발생할 수 있습니다.

2 이것은 FINAL 호출이며, 테이블 함수에 대하여 FINAL CALL이 지정된 경우에만 발생합니다. 이것은 FIRST 호출과 짝을 이루며, 명령문 실행시 단 한 번만 발생합니다. 이것은 자원 릴리스를 목적으로 합니다.

자원 릴리스

UDF를 작성하여 확보한 자원을 릴리스하십시오. 테이블 함수의 경우, 이 릴리스에는 CLOSE 호출과 FINAL 호출의 두 가지 발생 위치가 있습니다. CLOSE 호출은 각각의 OPEN 호출과 짝을 이루며, 명령문 실행시 여러번 발생할 수 있습니다. FINAL 호출은 UDF에 FINAL CALL이 지정되어 있는 경우 발생할 수 있으며, 명령문에서 한 번만 발생합니다.

UDF의 모든 OPEN/FETCH/CLOSE 절차에 자원을 적용할 수 있는 경우, FIRST 호출에서 자원을 확보하고 FINAL 호출에서 이것을 해제하도록 UDF를 작성하십시오. 스크래치 패드는 이 자원을 추적하기 위한 발생 위치입니다. 테이블 함수의 경우, FINAL CALL이 지정된 경우, FIRST 호출 이전에만 스크래치 패드가 초기화됩니다. FINAL CALL이 지정되지 않은 경우, 각 OPEN 호출 전에 다시 초기화됩니다.

자원이 OPEN/FETCH/CLOSE 절차에 고유한 경우, CLOSE 호출시 자원을 해제하도록 UDF를 작성하십시오. 테이블 함수가 부속 조회나 조인 내에 있는 경우, DB2 최적화 알고리즘이 명령문 실행을 조직화하기 위해 선택하는 방법에 따라, OPEN/FETCH/CLOSE가 여러 번 발생할 수 있습니다.

dbinfo 이 인수는 UDF를 호출하기 전에 DB2에 의해 설정됩니다. 이것은 UDF에 대한 CREATE FUNCTION문이 DBINFO 키워드를 지정한 경우에만 존재합니다. 인수는 465 페이지의『UDF include 파일: sqludf.h』에서 설명된 `sqludf.h` 헤더 파일에서 정의된 `sqludf_dbinfo` 구조입니다. 이 구조의 변수에는 이름이 포함되며, 식별자는 이 DB2 릴리스에서 허용되는 가장 긴 값보다 더 길 수 있으나, 차후의 릴리스와의 호환성을 위해 이

런 방법으로 정의됩니다. 실제 사용되는 변수의 일부를 읽거나 추출하기 위해 각 이름과 식별자 변수를 보완하는 길이 변수를 사용할 수 있습니다. *dbinfo* 구조에는 다음과 같은 요소들이 포함됩니다.

1. 데이터베이스 이름 길이(dbnamelen)

   *데이터베이스 이름*의 길이는 다음과 같습니다. 이 필드는 부호없는 short 정수입니다.

2. 데이터베이스 이름(dbname)

   현재 연결된 데이터베이스의 이름. 이 필드는 128자의 긴 식별자입니다. 위에서 설명한 *데이터베이스 이름 길이* 필드는 이 필드의 실제 길이를 식별합니다. 여기에는 널(NULL) 종료자나 패딩이 포함되지 않습니다.

3. 응용프로그램 권한 부여 ID 길이(authidlen)

   아래의 *응용프로그램 권한 부여 ID*의 길이. 이 필드는 부호없는 short 정수입니다.

4. 응용프로그램 권한 부여 ID(authid)

   응용프로그램 런타임 권한 부여 ID. 이 필드는 128자의 긴 식별자입니다. 여기에는 널(NULL) 종료자나 패딩이 포함되지 않습니다. 위에서 설명한 *응용프로그램 권한 부여 ID 길이* 필드는 이 필드의 실제 길이를 식별합니다.

5. 데이터베이스 코드 페이지(codepg)

   이것은 48 바이트 길이의 구조 두 개로 구성된 유니온으로 하나는 DB2 Universal Database에 의해 사용되고 다른 하나는 차후 사용을 위해 예약된 것입니다. DB2 Universal Database에 의해 사용되는 구조에는 다음과 같은 필드가 포함됩니다.

   a. SBCS. 1 바이트 코드 페이지, 부호없는 long 정수.
   b. DBCS. 1 바이트 코드 페이지, 부호없는 long 정수.
   c. COMP. 복합 코드 페이지, 부호없는 long 정수.

6. 스키마 이름 길이(tbschemalen)

   아래의 *스키마 이름*의 길이. 테이블 이름이 전달되지 않으면 0이 포함됩니다. 이 필드는 부호없는 short 정수입니다.

7. 스키마 이름(tbschema)

   아래의 *테이블 이름*에 대한 스키마. 이 필드는 128자의 긴 식별자입니다. 여기에는 널(NULL) 종료자나 패딩이 포함되지 않습니다. 위에서 설명한 *스키마 이름 길이* 필드는 이 필드의 실제 길이를 식별합니다.

8. 테이블 이름 길이(tbnamelen)

   아래의 *테이블 이름*의 길이. 테이블 이름이 전달되지 않으면 0이 포함됩니다. 이 필드는 부호없는 short 정수입니다.

9. 테이블 이름(tbname)

   이것은 갱신되거나 삽입되는 테이블의 이름입니다. 이 필드는 UDF 참조가 UPDATE문에서 SET절의 오른쪽에 있거나 INSERT문의 VALUES 목록의 항목인 경우에만 설정됩니다. 이 필드는 128자의 긴 식별자입니다. 여기에는 널(NULL) 종료자나 패딩이 포함되지 않습니다. 위에서 설명한 *테이블 이름 길이* 필드는 이 필드의 실제 길이를 식별합니다. 위의 *스키마 이름* 필드는 이 필드와 함께 완전한 테이블 이름을 구성합니다.

10. 컬럼 이름 길이(colnamelen)

    아래의 *컬럼 이름*의 길이. 컬럼 이름이 전달되지 않으면 0이 포함됩니다. 이 필드는 부호없는 short 정수입니다.

11. 컬럼 이름(colname)

    테이블 이름과 완전히 똑같은 조건 하에서, 이 필드는 갱신되거나 삽입되는 컬럼의 이름이 포함되며, 그렇지 않은 경우 예측할 수 없습니다. 이 필드는 128자의 긴 식별자입니다. 여기에는 널(NULL) 종료자나 패딩이 포함되지 않습니다. 위에서 설명한 *컬럼 이름 길이* 필드는 이 필드의 실제 길이를 식별합니다.

12. 버전/릴리스 번호(ver_rel)

    제품과 이것의 버전, 릴리스 및 수정 레벨을 식별하는 8자 필드로, 형식은 *pppvvrrm*이고 여기에서,

    - *ppp*는 다음과 같이 제품을 식별합니다.

      DSN MVS/ESA용 DB2 또는 OS/390

      ARI SQL/DS

QSQ AS/400용 DB2 Universal Database

SQL DB2 Universal Database

- *vv*는 두 자리 숫자의 버전 식별자.
- *rr*은 두 자리 숫자의 릴리스 식별자.
- *m*은 한 자리 숫자의 수정 레벨 식별자.

13. 플랫폼(platform)

    응용프로그램 서버에 대한 운영 플랫폼은 다음과 같습니다.

    SQLUDF_PLATFORM_AIX
    AIX

    SQLUDF_PLATFORM_HP HP-UX

    SQLUDF_PLATFORM_MVS
    OS/390

    SQLUDF_PLATFORM_NT Windows NT

    SQLUDF_PLATFORM_OS2
    OS/2

    SQLUDF_PLATFORM_SUN
    Solaris Operating Environment

    SQLUDF_PLATFORM_WINDOWS
    Windows 95 및 Windows 98

    SQLUDF_PLATFORM_UNKNOWN
    알 수 없는 플랫폼

    위 목록에 없는 기타 플랫폼의 경우, `sqludf.h` 파일을 참조하십시오.

14. 테이블 함수 컬럼 항목(numtfcol)의 번호

    아래의 *테이블 함수 컬럼 목록*에 지정된 테이블 함수 컬럼 목록의 0이 아닌 항목의 수.

15. 예약된 필드(resd1)

    이 필드는 차후 사용을 위한 것입니다. 이것은 24 문자 길이로 정의됩니다.

16. 테이블 함수 컬럼 목록(tfcolumn)

이것이 테이블 함수인 경우, 이 필드는 DB2에 의해 동적으로 할당되는 short 정수 배열에 대한 포인터입니다. 스칼라 함수인 경우, 이 포인터는 널(NULL)입니다.

이 필드는 테이블 함수에 대해서만 사용됩니다. *n*이 *테이블 함수 컬럼 목록 항목 수* 필드인 numtfcol에 지정되어 있는 첫번째 *n* 항목만이 관계가 있습니다. *n*은 0이거나 CREATE FUNCTION문의 RETURNS TABLE(...)절의 함수에 정의된 결과 컬럼의 수와 같거나 이보다 작습니다. 이 값들은 이 명령문이 테이블 함수로부터 필요로 하는 컬럼의 순번에 대응됩니다. 즉, '1' 값은 첫번째로 정의된 결과 컬럼을 의미하며, '2'는 두번째로 정의된 결과 컬럼을 의미하고, 이 값은 어떤 순서도 가능합니다. *n*이 0일 수 있음에 주의하십시오. 즉, 조회가 실제 컬럼 값을 필요로 하지 않는 SELECT COUNT(*) FROM TABLE(TF(...)) AS QQ와 유사한 명령문에 대하여 변수 numtfcol이 0일 수도 있습니다.

이 배열은 최적화에 대한 기회를 나타냅니다. UDF는 테이블 함수의 모든 결과 컬럼에 대한 모든 값을 리턴할 필요가 없으며, 특정 문맥에서 필요로 하는 값들만을 리턴하며, 이것은 배열에서 숫자에 의해 식별되는 컬럼입니다. 이 최적화가 성능을 향상시키기 위해 UDF 논리를 복잡하게 할 수 있으므로, UDF는 모든 계산 결과 컬럼을 리턴하는 것으로 선택할 수 있습니다.

17. 고유 응용프로그램 식별자(appl_id)

    이 필드는 응용프로그램과 DB2의 연결을 고유하게 식별하는 C의 널(NULL) 종료 문자열에 대한 포인터입니다. 이것은 각각의 데이터베이스 연결시 다시 생성됩니다.

    문자열은 최대 32자 길이이며, 정확한 형식은 클라이언트와 DB2간에 설정된 연결 유형에 따라 다릅니다. 일반적으로 형식은 다음과 같습니다.

```
<x>.<y>.<ts>
```

    여기에서 <x>와 <y>는 연결 유형에 따라 다르지만, <ts>는 12자의 YYMMDDHHMMSS 형식의 시간소인으로 잠재적으로 DB2에 의해 고유한 상태를 유지하도록 조정됩니다.

예: *LOCAL.db2inst.980707130144

18. 예약된 필드(resd2)

    이 필드는 차후 사용을 위한 것입니다. 이것은 20 문자 길이로 정의됩니다.

UDF 인수 사용 요약

다음은 위에서 설명된 인수들에 대한 요약 및 DB2와 외부 UDF간의 인터페이스에서 이들을 사용하는 방법입니다.

스칼라 함수의 경우 인수는 다음과 같습니다.

- *SQL-argument*.

  이 인수는 함수 참조에서 식별된 값을 DB2에서 UDF로 전달합니다. 각 SQL 인수에 이들 인수들 중 하나가 있습니다.

- *SQL-result*.

  이 인수는 UDF에 의해 생성된 결과 값을 DB2 및 함수 참조가 발생하는 SQL 문으로 전달합니다.

- *SQL-argument-ind*.

  이 인수는 위치에 있어서 *SQL-argument*에 대응하고, 특정 인수가 널(NULL)인지 여부를 UDF에 알려줍니다. 각 *SQL-argument*에 이들 인수들 중 하나가 있습니다.

- *SQL-result-ind*.

  이 인수는 함수 결과 *SQL-result*에 널(NULL)이 들어 있는지 여부를 다시 DB2에 보고하기 위해 UDF가 사용합니다.

- *SQL-state* 및 *diagnostic-message*.

  이 인수들은 예외 정보를 다시 DB2로 알리기 위해 UDF에 의해 사용됩니다.

- *function-name* 및 *specific-name*.

  이 인수들은 참조된 함수의 식별자를 UDF로 전달하는 데 사용됩니다.

- *scratchpad* 및 *call-type*.

이 인수들은 호출과 호출 사이에서 UDF 상태 저장을 관리하는 데 DB2에 의해 사용됩니다. DB2에 의해 *스크래치 패드*가 작성되고 *초기화*되며, UDF가 이를 관리합니다. DB2는 *call-type* 인수를 사용하여 UDF로 호출 유형을 신호합니다.

- *dbinfo*.

  DB2에 의해 UDF로 전달되는, 추가 정보를 담고 있는 구조.

테이블 함수는 논리적으로 이것을 참조하는 SQL문으로 테이블을 리턴하지만, DB2와 테이블 함수간의 실제 인터페이스는 행 단위입니다. 테이블 함수의 경우, 인수는 다음과 같습니다.

- *SQL-argument*.

  이 인수는 함수 참조에서 식별된 값을 DB2에서 UDF로 전달합니다. 이 인수는 OPEN 및 FIRST 호출에서와 같이 FETCH 호출에 대하여 동일한 값을 갖습니다. 각 SQL 인수에 이들 중 하나가 있습니다.

- *SQL-result*.

  이 인수는 UDF에 의해 리턴되는 행에 대한 각 컬럼 값을 다시 전달하는 데 사용됩니다. CREATE FUNCTION문의 RETURNS TABLE (...) 절에 정의된 각 결과 컬럼 값에 대하여 이들 인수들 중 하나가 작습니다.

- *SQL-argument-ind*.

  이 인수는 위치에 있어서 *SQL-argument* 값에 대응하고, 특정 인수가 널(NULL)인지 여부를 UDF에게 알려줍니다. 각 SQL 인수에 이들 중 하나가 있습니다.

- *SQL-result-ind*.

  이 인수는 테이블 함수 출력 행에 리턴된 개별적인 컬럼 값이 널(NULL)인지 여부를 다시 DB2에 보고하기 위해 UDF에 의해 사용됩니다. 이것은 위치에 있어서 *SQL-result* 인수에 대응됩니다.

- *SQL-state* 및 *diagnostic-message*.

  이 인수는 예외 정보 및 테이블 끝 조건을 다시 DB2로 신호하기 위해 UDF에 의해 사용됩니다.

- *function-name* 및 *specific-name*.

  이 인수들은 참조된 함수의 식별자를 UDF로 전달하는 데 사용됩니다.

- *scratchpad* 및 *call-type*.

  이 인수들은 호출과 호출 사이에서 UDF 상태 저장을 관리하는 데 DB2에 의해 사용됩니다. DB2에 의해 *스크래치 패드*가 작성되고 *초기*화되며, UDF가 이를 관리합니다. DB2는 *call-type* 인수를 사용하여 UDF로 호출 유형을 신호합니다. 테이블 함수의 경우 이들 호출 유형은 OPEN, FETCH, CLOSE이며, 옵션은 FIRST와 FINAL입니다.

- *dbinfo*.

  이것은 추가 정보를 담고 있는, DB2에 의해 UDF로 전달되는 구조입니다.

DB2에서 UDF로 전달되는 인수를 사용하여 *SQL-result*, *SQL-result-ind* 및 *SQL-state* 뿐만 아니라 정상적인 UDF의 값 출력이 DB2로 리턴되는 것을 지켜보십시오. 사실 UDF는 함수적 의미에서는 아무것도 리턴하지 않도록 작성됩니다. 즉, 함수의 리턴 유형은 void입니다. 다음 예에서 `void` 정의와 `return`문을 참조하십시오.

```
#include ...
 void SQL_API_FN divid(
      ... arguments ... )
{
      ... UDF body ...
      return;
}
```

위의 예에서 `SQL_API_FN`은 지원되는 운영 체제에 따라 다를 수 있는 함수에 대한 호출 규약을 지정하는 마크로입니다. 이 마크로는 저장 프로시듀어나 UDF를 작성할 때 필요합니다.

UDF의 프로그래밍 예는 505 페이지의 『UDF 코드 예』의 내용을 참조하십시오.

SQL 데이터 유형이 UDF로 전달되는 방법

이 절에서는 UDF의 매개변수와 결과 모두에 유효한 유형을 식별하고, 해당 인수가 사용자의 C 또는 C++ 언어의 UDF에서 정의되는 방식을 각각 지정합니다. Java UDF에서의 유형 정의는 712 페이지의 『Java에서 지원되는 SQL 데이터 유형』의 내용을 참조하십시오. 사용자가 `sqludf.h` include 파일 및 여기에 정의된 유형을 사용하는 경우, 서로 다른 데이터 유형 및 컴파일러에 올바른 언어 변수와 구조를

자동으로 생성할 수 있습니다. 예를 들어, BIGINT의 경우, SQLUDF_BIGINT 데이터 유형을 사용하여 서로 다른 컴파일러간에 64비트 정수 유형의 이름으로 차이를 숨길 수 있습니다. 이 include 파일은 465 페이지의 『UDF include 파일: sqludf.h』에서 논의됩니다.

이것은 인수 값의 형식을 제어하는, CREATE FUNCTION문에 정의된 각 함수 매개변수에 대한 데이터 유형입니다. 이 형식의 값을 구하려면, 인수 데이터 유형으로부터의 승격이 필요할 수 있습니다. 이러한 승격은 DB2에 의해 인수 값에서 자동으로 수행됩니다. 인수 승격은 *SQL 참조서*에서 설명됩니다.

함수 결과의 경우, 이것은 형식을 정의하는 CREATE FUNCTION문의 CAST FROM절에 지정된 데이터 유형입니다. CAST FROM절이 없는 경우, RETURNS 절에 지정된 데이터 유형이 형식을 정의합니다.

다음 예에서, CAST FROM절이 존재한다는 것은 UDF 본문이 SMALLINT를 리턴함을 의미하며, DB2가 이 값을 함수 참조가 발생하는 명령문으로 전달하기 전에 INTEGER로 유형변환함을 의미합니다.

```
... RETURNS INTEGER CAST FROM SMALLINT ...
```

이 경우 아래에 정의된 대로 SMALLINT를 생성하기 위해 UDF를 작성해야 합니다. CAST FROM 데이터 유형은 RETURNS 데이터 유형으로 *유형변환 가능*하게 하여 임의로 다른 데이터 유형을 선택할 수 없도록 해야 합니다. 데이터 유형간 유형변환에 대해서는 *SQL 참조서*에서 설명합니다.

다음은 SQL 유형 및 이들의 C 언어에서의 표기 목록입니다. Java에서의 SQL 유형 표기 목록은 712 페이지의 『Java에서 지원되는 SQL 데이터 유형』의 내용을 참조하십시오. 여기에는 각 유형이 매개변수나 결과로서 유효한지 여부에 대한 정보가 포함됩니다. 또한 사용자의 C 또는 C++ 언어 UDF에서의 인수 정의로 유형이 나타날 수 있는 방법의 예도 포함됩니다.

- SMALLINT

  유효함. C 언어에서는 `short`로 표시합니다.

정수 UDF 매개변수를 정의할 때, DB2가 SMALLINT 인수를 INTEGER로 승격하지 않으므로, SMALLINT보다는 INTEGER 사용을 고려하십시오. 예를 들어, UDF를 다음과 같이 정의한다고 가정하십시오.

```
CREATE FUNCTION SIMPLE(SMALLINT)...
```

예:

```
short    *arg1;          /* example for SMALLINT */
short    *arg1_null_ind; /* example for any null indicator */
```

INTEGER 데이터를 사용하여 SIMPLE 함수를 호출하는 경우 (... SIMPLE(1)...), 해당 함수를 찾을 수 없다는 오류 메시지 SQLCODE -440(SQLSTATE 42884)이 표시되며, 이 함수의 일반 사용자는 이 메시지의 원인을 알 수 없습니다. 위의 예에서 1은 INTEGER이므로, 이를 SMALLINT로 유형변환하거나 매개변수를 INTEGER로 정의할 수 있습니다.

- INTEGER 또는 INT

  유효함. C에서 sqlint32로 표시합니다. DB2 Include 파일인 sqlsystm.h는 이 유형을 플랫폼에 적합한 32비트 정수로 정의합니다.

  예:

```
sqlint32 *arg2;          /* example for INTEGER */
```

- BIGINT

  유효함. C 언어에서는 sqlint64로 표시합니다.

  예:

```
sqlint64 *arg3;          /* example for INTEGER */
```

  DB2는 sqlint64 C 언어 유형을 정의하여 컴파일러와 운영 체제에서 64비트 부호화 정수에 대한 정의의 차이를 극복합니다. 정의를 지정하려면 #include sqludf.h를 포함시켜야 합니다.

- DECIMAL(p,s) 또는 NUMERIC(p,s)

  C 언어 표기를 할 수 없으므로 **유효하지 않습니다.** 십진수 값을 전달하려는 경우, 매개변수를 DECIMAL로부터 유형변환 가능한 데이터 유형(예를 들어,

CHAR 또는 DOUBLE)으로 지정해야 하고, 해당 유형에 인수를 명시적으로 유형변환해야 합니다. DOUBLE의 경우, DB2가 자동으로 승격시키므로 십진수 인수를 DOUBLE 매개변수로 명시적으로 유형변환할 필요가 없습니다.

예를 들어, DECIMAL(5,2)로 WAGE 및 DECIMAL(4,1)로 HOURS라는 두 개의 컬럼이 있고, 임금, 노동 시간 및 기타 다른 인수에 근거하여 주급을 계산하는 UDF를 작성한다고 가정하십시오. UDF는 다음과 같습니다.

```
CREATE FUNCTION WEEKLY_PAY (DOUBLE, DOUBLE, ...)
       RETURNS DECIMAL(7,2) CAST FROM DOUBLE
       ...;
```

위의 UDF에서 첫번째 두 개의 매개변수는 임금과 시간 수에 해당됩니다. SQL 선택 명령문에서 UDF WEEKLY_PAY를 다음과 같이 호출합니다.

```
SELECT WEEKLY_PAY (WAGE, HOURS, ...) ...;
```

DECIMAL 인수는 DOUBLE로 유형변환될 수 있으므로, 명시적 유형변환은 필요하지 않습니다.

또다른 방법으로 다음과 같이 CHAR 인수를 사용하여 WEEKLY_PAY를 정의할 수 있습니다.

```
CREATE FUNCTION WEEKLY_PAY (VARCHAR(6), VARCHAR(5), ...)
       RETURNS DECIMAL (7,2) CAST FROM VARCHAR(10)
       ...;
```

다음과 같이 호출할 수 있습니다.

```
SELECT WEEKLY_PAY (CHAR(WAGE), CHAR(HOURS), ...) ...;
```

DECIMAL 인수가 VARCHAR로 승격될 수 없으므로 명시적 유형변환이 필요합니다.

부동 소수점 매개변수 사용의 이점은 UDF 값에 산술 연산을 수행하는 것이 더 쉽다는 점입니다. 문자 매개변수 사용의 이점은 항상 정확하게 십진수 값을 표시할 수 있다는 것입니다. 이것은 부동 소수점으로는 항상 가능하지는 않습니다.

- REAL

  유효함. C 언어에서는 `float`로 표시합니다.

예:

```
float *result;          /* example for REAL */
```

- DOUBLE 또는 DOUBLE PRECISION 또는 FLOAT

  유효함. C 언어에서는 double로 표시합니다.

  예:

```
double  *result;        /* example for DOUBLE   */
```

- FOR BIT DATA 수정자를 사용하거나 사용하지 않는 CHAR(n) 또는 CHARACTER(n).

  유효함. C에서는 char...[n+1]로 표시됩니다. 이것은 C의 널(NULL) 종료 문자열로 마지막 문자는 널(NULL)인 X'00'입니다.

  예:

```
char     arg1[14];      /* example for CHAR(13)   */
char    *arg1;          /* also perfectly acceptable */
```

CHAR(n) 매개변수의 경우, DB2는 항상 n 바이트의 데이터를 버퍼로 이동시키고, *n+1* 바이트를 널(NULL)로 설정합니다. RETURNS CHAR(n) 값의 경우, DB2는 항상 n 바이트를 갖고 *n+1* 바이트를 무시합니다. 이러한 RETURNS CHAR(n)의 경우, 처음 n 문자에 널(NULL) 문자를 잘못 입력하면 경고가 발생합니다. DB2는 이것을 정상적인 데이터의 일부가 아닌 다른 것으로 인식하지는 않으며, 이것이 나중에 의도하지 않은 이상한 결과를 발생시킬 수도 있습니다.

FOR BIT DATA가 지정된 경우, UDF에서 함수를 처리하는 정상적인 C 문자열을 사용하는 것에 주의해야 합니다. 이러한 함수들 중 다수가 문자열을 구분하는 널(NULL)을 탐색하며, 널(NULL) 문자(X'00')는 데이터 값의 중간에 포함될 수 있는 적절한 문자일 수 있습니다.

UDF 매개변수를 정의할 때, DB2가 VARCHAR 인수를 CHAR로 승격시키지 않으므로, CHAR보다는 VARCHAR을 사용하는 것을 고려하십시오. 예를 들어, UDF를 다음과 같이 정의하십시오.

```
CREATE FUNCTION SIMPLE(INT,CHAR(1))...
```

VARCHAR 데이터를 사용하여 SIMPLE 함수를 호출하는 경우 (... SIMPLE(1,'A')...), 해당 함수를 찾을 수 없다는 오류 메시지 SQLCODE -440(SQLSTATE 42884)이 표시되며, 이 함수의 일반 사용자는 이 메시지의 원인을 알 수 없습니다. 위의 예에서 'A'는 VARCHAR이므로, 이것을 CHAR로 유형변환하거나 매개변수를 VARCHAR로 정의할 수 있습니다.

- FOR BIT DATA 수정자를 사용하거나 사용하지 않는 VARCHAR(n) FOR BIT DATA 또는 LONG VARCHAR.

  유효함. C에서는 다음과 유사한 구조로 표시됩니다.

```
struct sqludf_vc_fbd
{
   unsigned short length;         /* length of data */
   char           data[1];        /* first char of data */
};
```

  [1]은 단지 컴파일러 배열을 지정할 뿐입니다. 이것은 단 하나의 문자만이 전달됨을 의미하지는 않습니다. 실제 구조가 아닌 이 구조의 주소가 전달되므로, 배열 논리를 사용할 수 있는 방법을 제공하는 것입니다.

  널(NULL) 문자가 정상적인 데이터 값의 일부가 될 수 있으므로, 이 값들은 C의 널(NULL) 종료 문자열로 표시되지 않습니다. 길이는 구조 변수 length를 사용하는 매개변수에 대한 UDF로 명시적으로 전달됩니다. RETURNS절의 경우, UDF로 전달되는 길이는 버퍼의 길이입니다. UDF 본문이 구조 변수 length를 사용하여 다시 전달해야 하는 것은 데이터 값의 실제 길이입니다.

  예:

```
struct sqludf_vc_fbd *arg1;  /* example for VARCHAR(n) FOR BIT DATA */
struct sqludf_vc_fbd *result; /* also for LONG VARCHAR FOR BIT DATA */
```

- FOR BIT DATA를 사용하지 않는 VARCHAR(n).

  유효함. C 언어에서는 char...[n+1]로 표시합니다. 이것은 C의 널(NULL) 종료 문자열입니다.

  VARCHAR(n) 매개변수의 경우, DB2는 (k+1) 위치에 널(NULL)을 넣는데, 여기서 k는 특정 발생 길이입니다. C 문자열 처리 함수는 이렇게 이들 값의 조작에 매우 적절합니다. RETURNS VARCHAR(n) 값의 경우, UDF 본문은 널

(NULL)을 사용하여 실제 값을 구분해야 하는데, 이는 DB2가 이 널(NULL) 문자로 결과 길이를 판별하기 때문입니다.

예:

```
char     arg2[51];       /* example for VARCHAR(50)   */
char    *result;         /* also perfectly acceptable */
```

- GRAPHIC(n)

  유효함. C 언어에서는 `sqldbchar[n+1]`로 표시합니다. 이것은 널(NULL) 종료 그래픽 문자열입니다. `wchar_t[n+1]`은 `wchar_t`가 길이 2 바이트로 정의되는 플랫폼에서 사용할 수 있습니다. 그러나 `sqldbchar`이 권장됩니다. 이 두 가지 데이터 유형에 대한 정보는 695 페이지의 『C 및 C++에서 wchar_t 또는 sqldbchar 데이터 유형 선택』의 내용을 참조하십시오.

  GRAPHIC(n) 매개변수의 경우, DB2는 n개의 2 바이트 문자들을 버퍼로 이동하고, 다음의 두 바이트를 널(NULL)로 설정합니다. DB2에서 UDF로 전달된 데이터는 DBCS 형식이며, 다시 전달된 결과도 DBCS 형식이어야 합니다. 이러한 형태는 696 페이지의 『C 및 C++의 WCHARTYPE 사전 처리 컴파일러 옵션』에서 설명된 WCHARTYPE NOCONVERT 사전 처리 컴파일러를 사용하는 것과 같습니다. RETURNS GRAPHIC(n) 값의 경우, DB2는 항상 n개의 2 바이트 문자를 취하고 다음의 바이트들을 무시합니다.

  그래픽 UDF 매개변수를 정의할 때, DB2가 VARGRAPHIC 인수를 GRAPHIC으로 승격시키지 않으므로 GRAPHIC보다는 VARGRAPHIC 사용을 고려하십시오. 예를 들어 UDF를 다음과 같이 정의한다고 가정합니다.

  ```
  CREATE FUNCTION SIMPLE(GRAPHIC)...
  ```

  VARGRAPHIC 데이터를 사용하여 SIMPLE 함수를 호출하는 경우 (`... SIMPLE('`*`graphic_literal`*`')...`), 사용자에게는 해당 함수를 찾을 수 없다는 SQLCODE -440(SQLSTATE 42884) 오류 메시지가 표시되며, 이 함수의 최종 사용자는 이 메시지의 원인을 이해할 수 없을 것입니다. 위의 예에서, *`graphic_literal`*은 VARGRAPHIC 데이터로 해석되는 리터럴 DBCS 문자열이므로, GRAPHIC으로 유형변환하거나 매개변수를 VARGRAPHIC으로 정의하십시오.

  예:

```
sqldbchar   arg1[14];       /* example for GRAPHIC(13)   */
sqldbchar  *arg1;           /* also perfectly acceptable */
```

- VARGRAPHIC(n)

  유효함. C 언어에서는 sqldbchar[n+1]로 표시합니다. 이것은 널(NULL) 종료 그래픽 문자열입니다. wchar_t[n+1]은 wchar_t가 길이 2 바이트로 정의되는 플랫폼에서 사용할 수 있습니다. 그러나 sqldbchar이 권장됩니다. 이 두 가지 데이터 유형에 대한 정보는 695 페이지의 『C 및 C++에서 wchar_t 또는 sqldbchar 데이터 유형 선택』의 내용을 참조하십시오.

  VARGRAPHIC(n) 매개변수의 경우, DB2는 그래픽 널(NULL)을 (k+1) 위치에 넣는데, 여기에서 k는 특정 발생 길이입니다. 그래픽 널(NULL)이란 그래픽 문자열의 마지막 문자의 모든 바이트에 2진 0('\0's)이 들어 있는 조건을 말합니다. DB2에서 UDF로 전달된 데이터는 DBCS 형식이며, 다시 전달된 결과도 DBCS 형식이어야 합니다. 이러한 형태는 696 페이지의 『C 및 C++의 WCHARTYPE 사전 처리 컴파일러 옵션』에서 설명된 WCHARTYPE NOCONVERT 사전 처리 컴파일러를 사용하는 것과 같습니다. RETURNS VARGRAPHIC(n) 값의 경우, UDF 본문은 그래픽 널(NULL)을 사용하여 실제 값을 구분하는데, 이는 DB2가 이 그래픽 널(NULL) 문자로 결과 길이를 판별하기 때문입니다.

  예:

  ```
  sqldbchar   args[51],     /* example for VARGRAPHIC(50) */
  sqldbchar   *result,      /* also perfectly acceptable  */
  ```

- LONG VARGRAPHIC

  유효함. C 언어에서는 struct로 표시됩니다.

  ```
  struct sqludf_vg
  {
     unsigned short length;          /* length of data */
     sqldbchar      data[1];         /* first char of data */
  };
  ```

  위의 구조에서 wchar_t가 길이 2 바이트로 정의되는 플랫폼에서 sqldbchar 대신 wchar_t를 사용할 수 있으나, sqldbchar 사용이 권장됩니다. 이 두 가지 데이터 유형에 대한 정보는 695 페이지의 『C 및 C++에서 wchar_t 또는 sqldbchar 데이터 유형 선택』의 내용을 참조하십시오.

[1]은 단지 컴파일러에게 배열을 가리킵니다. 이것이 하나의 그래픽 문자만이 전달됨을 의미하는 것은 아닙니다. 실제 구조가 아닌 이 구조의 주소가 전달되므로, 배열 논리를 사용할 수 있는 방법을 제공하는 것입니다.

이것은 널(NULL) 종료 그래픽 문자열로 표시되지 않습니다. 2 바이트 문자에서 길이는 구조 변수 length를 사용하는 매개변수에 대한 UDF로 명시적으로 전달됩니다. DB2에서 UDF로 전달된 데이터는 DBCS 형식이며, 다시 전달된 결과도 DBCS 형식이어야 합니다. 이러한 형태는 696 페이지의 『C 및 C++의 WCHARTYPE 사전 처리 컴파일러 옵션』에서 설명된 WCHARTYPE NOCONVERT 사전 처리 컴파일러를 사용하는 것과 같습니다. RETURNS절의 경우, UDF로 전달되는 길이는 버퍼의 길이입니다. UDF 본문이 구조 변수 length를 사용하여 다시 전달해야 하는 것은 2 바이트 문자에서의 실제 데이터 값 길이입니다.

예:

```
struct sqludf_vg *arg1;  /* example for VARGRAPHIC(n)   */
struct sqludf_vg *result; /* also for LONG VARGRAPHIC   */
```

- DATE

  유효함. C에서 CHAR(10), 즉 char...[11]과 동일하게 표시됩니다. 날짜 값은 항상 ISO 형식인 yyyy-mm-dd로 UDF에 전달됩니다.

  예:

```
char     arg1[11];      /* example for DATE        */
char    *result;        /* also perfectly acceptable */
```

- TIME

  유효함. C에서 CHAR(8), 즉 char...[9]와 동일하게 표시됩니다. 시간 값은 항상 ISO 형식인 hh.mm.ss로 UDF에 전달됩니다.

  예:

```
char    *arg;           /* example for DATE           */
char     result[9];     /* also perfectly acceptable */
```

- TIMESTAMP

  유효함. C에서 CHAR(26)인 char...[27]과 동일하게 표시됩니다. 시간소인 값은 항상 yyyy-mm-dd-hh.mm.ss.nnnnnn 형식으로 전달됩니다.

예:

```
char     arg1[27];      /* example for TIMESTAMP  */
char    *result;        /* also perfectly acceptable */
```

- BLOB(n) 및 CLOB(n)

  유효함. C 언어에서는 struct로 표시됩니다.

```
struct sqludf_lob
{
    sqluint32     length;      /* length in bytes */
    char          data[1];      /* first byte of lob */
};
```

  [1]은 단지 컴파일러에게 배열을 가리킵니다. 이것은 단 하나의 문자만이 전달됨을 의미하지는 않습니다. 실제 구조가 아닌 이 구조의 주소가 전달되므로, 배열 논리를 사용할 수 있는 방법을 제공하는 것입니다.

  이것은 널(NULL) 종료 문자열로 표시되지 않습니다. 길이는 구조 변수 length를 사용하는 매개변수에 대한 UDF로 명시적으로 전달됩니다. RETURNS절의 경우, UDF로 전달되는 길이는 버퍼의 길이입니다. UDF 본문이 구조 변수 length를 사용하여 다시 전달해야 하는 것은 데이터 값의 실제 길이입니다.

  예:

```
struct sqludf_lob *arg1;  /* example for BLOB(n), CLOB(n) */
struct sqludf_lob *result;
```

- DBCLOB(n)

  유효함. C 언어에서는 struct로 표시됩니다.

```
struct sqludf_lob
{
    sqluint32 length;      /* length in graphic characters */
    sqldbchar data[1];         /* first byte of lob */
};
```

  위의 구조에서 wchar_t가 길이 2 바이트로 정의되는 플랫폼에서 sqldbchar 대신 wchar_t를 사용할 수 있으나, sqldbchar 사용이 권장됩니다. 이 두 가지 데이터 유형에 대한 정보는 695 페이지의 『C 및 C++에서 wchar_t 또는 sqldbchar 데이터 유형 선택』의 내용을 참조하십시오.

[1]은 단지 컴파일러에게 배열을 가리킵니다. 이것은 단 하나의 그래픽 문자만 이 전달됨을 의미하지는 않습니다. 실제 구조가 아닌 이 구조의 주소가 전달되므로, 배열 논리를 사용할 수 있는 방법을 제공하는 것입니다.

이것은 널(NULL) 종료 그래픽 문자열로 표시되지 않습니다. 길이는 구조 변수 length를 사용하는 매개변수에 대한 UDF로 명시적으로 전달됩니다. DB2에서 UDF로 전달된 데이터는 DBCS 형식이며, 다시 전달된 결과도 DBCS 형식이어야 합니다. 이러한 형태는 696 페이지의 『C 및 C++의 WCHARTYPE 사전 처리 컴파일러 옵션』에서 설명된 WCHARTYPE NOCONVERT 사전 처리 컴파일러를 사용하는 것과 같습니다. RETURNS절의 경우, UDF로 전달되는 길이는 버퍼의 길이입니다. UDF 본문이 구조 변수 length를 사용하여 다시 전달해야 하는 것은 2 바이트 문자에서 표현되는 이들 길이 모두를 사용하여 실제 데이터 값의 길이입니다.

예:

```
struct sqludf_lob *arg1;  /* example for DBCLOB(n) */
struct sqludf_lob *result;
```

- 구별 유형

기본 유형에 따라 유효하거나 유효하지 않음. 구별 유형은 UDT의 기본 유형 형식으로 UDF에 전달되므로, 기본 유형이 유효하거나, 그러한 경우에만 지정될 수 있습니다.

예:

```
struct sqludf_lob *arg1;  /* for distinct type based on BLOB(n) */
double            *arg2;  /* for distinct type based on DOUBLE  */
char              res[5]; /* for distinct type based on CHAR(4) */
```

- 구별 유형 AS LOCATOR 또는 기타 LOB 유형 AS LOCATOR

AS LOCATOR 유형 수정자는 UDF 매개변수와 결과 정의에서만 유효합니다. 이것은 LOB 유형이나 LOB 유형에 근거하는 구별 유형을 수정하는 데에만 사용할 수 있습니다. 유형 수정자를 지정하는 경우, 전체 LOB 값이 아닌 4 바이트의 위치 지정자가 UDF로 전달됩니다.

예:

```
sqludf_locator       *arg1;  /* locator argument */
sqludf_locator       *result; /* locator result */
```

udf_locator 유형은 465 페이지의 『UDF include 파일: sqludf.h』에서 설명되는 헤더 파일 sqludf.h에 정의됩니다. 이러한 위치 지정자 사용에 대한 내용은 493 페이지의 『UDF 매개변수 또는 결과로 LOB 위치 지정자 사용』에서 설명됩니다.

32-비트 및 64-비트 플랫폼에서의 스크래치 패드 작성

UDF 코드를 32-비트와 64-비트 플랫폼간에 이동 가능하도록 만들려면 64-비트 값이 들어 있는 스크래치 패드를 작성하고 사용하는 방법을 변경해야 합니다. 64-비트 포인터 또는 sqlint64 BIGINT 변수와 같은 64-비트 값을 하나 이상 포함하는 스크래치 패드 수도에 대해 길이 변수를 명시적으로 선언하지 않아도 됩니다. 예를 들어, 다음 예는 구조 선언에 명시적 길이 변수가 들어 있으므로 64-비트 플랫폼에서의 데이터 정렬 예외가 발생할 수 있습니다.

```
struct scratch1
{
  sqlint32 length;
  char chars[4];
  sqlint64 bigint_var;
};
```

32-비트와 64-비트 플랫폼간에 이동할 수 있도록 이전 예에서 스크래치 패드를 선언하려면 구조의 명시적 길이 변수의 선언을 제거하십시오. 다음 예는 명시적 길이 변수를 선언하지 않고 스크래치 패드 구조를 선언합니다.

```
struct scratch1
{
  sqlint64 bigint_var;
  char chars[4];
};
```

UDF에서 명시적 길이 변수를 선언하지 않는 스크래치 패드 구조를 액세스하려면 다음 형식을 사용하여 스크래치 패드를 참조할 수 있습니다.

```
struct scratchpad_data * data =
  (struct scratchpad_data*)scratch_pointer->data;
```

여기서 *scratch_pointer*는 UDF의 *sqludf_scratchpad* 포인터를 나타내고 *data*는 스크래치 패드의 내용을 나타냅니다.

UDF include 파일: sqludf.h

이 include 파일에는 UDF 작성시 유용한 구조, 정의 및 값들이 들어 있습니다. 이 파일을 사용하는 것은 선택적이지만, 505 페이지의『UDF 코드 예』의 UDF 예에서 일부 예는 이 include 파일을 사용합니다. UDF 컴파일시 이 파일이 들어 있는 디렉토리를 참조해야 합니다. 이 디렉토리는 `sqllib/include`입니다.

`sqludf.h` include 파일은 자기 서술적입니다. 다음은 그 내용을 간략하게 요약한 것입니다.

1. 구조인 전달된 인수에 대한 구조 정의는 다음과 같습니다.
   - VARCHAR FOR BIT DATA 인수 및 결과
   - LONG VARCHAR(FOR BIT DATA가 있거나 또는 없음) 인수 및 결과
   - LONG VARGRAPHIC 인수 및 결과
   - 모든 LOB 유형, SQL 인수 및 결과
   - 스크래치 패드
   - dbinfo 구조.
2. 모든 SQL 데이터 유형을 갖고 있는 SQL 인수 및 결과에 대응하는 UDF 인수 정의에 사용될, 모든 SQL 데이터 유형에 대한 C 언어 유형 정의. 이들은 SQLUDF_x and SQLUDF_x_FBD라는 이름을 갖는 정의인데, 여기에서 x는 SQL 데이터 유형 이름이고, FBD는 2진 데이터용(For Bit Data)을 나타냅니다.

   또한 AS LOCATOR 보조 프로그램을 사용하여 정의된 인수나 결과의 C 언어 유형이 포함됩니다.
3. *call-type* 인수의 `enum` 유형 정의를 사용한 *scratchpad*와 *call-type*인수에 대한 C 언어 유형 정의.
4. *scratchpad*와 *call-type* 인수를 포함하거나 또는 포함하지 않는 표준 *trailing* 인수 정의를 위한 마크로 이것은 함수 정의의 SCRATCHPAD와 FINAL CALL 키워드의 존재 여부에 대응합니다. 이들은 435 페이지의『인수를 DB2에서 UDF로 전달』에 정의된 *SQL-state*, *function-name*, *specific-name*,

diagnostic-message, *scratchpad* 및 *call-type* UDF 호출 인수입니다. 또한 이 구조물들과 여러가지 유효한 SQLSTATE 값을 참조하기 위한 정의들도 포함되어 있습니다.

5. SQL 인수가 널(NULL)인지 여부를 테스트하는 마크로.
6. UDF로 전달된 LOB 위치 지정자를 사용하여 LOB 값을 조작하는 데 사용될 수 있는 API에 대한 함수 프로토타입.

다음 절의 UDF 예 중 일부에서는 `sqludf.h`를 포함 및 사용하고 있습니다.

Java 사용자 정의 함수 작성 및 사용

몇가지 차이점을 제외하고는 다른 언어에서와 마찬가지로 Java로도 UDF를 작성하고 사용할 수 있습니다. UDF를 코딩한 후, CREATE FUNCTION문을 사용하여 이를 데이터베이스에 등록해야 합니다. 이 명령문을 사용하여 Java UDF를 등록하는 것에 대한 정보는 *SQL 참조서*에서 자세한 내용을 참조하십시오. 그리고 나서 응용프로그램의 SQL에서 이를 참조할 수 있습니다. UDF는 FENCED 또는 NOT FENCED 될 수 있으며, UDF 수행 방식을 수정하는 옵션을 사용할 수도 있습니다. 468 페이지의『Java UDF 실행 방법 변경』의 내용을 참조하십시오.

일부 샘플 Java UDF 본문이 UDFsrv.java 샘플에서 제공됩니다. 연관된 CREATE FUNCTION문과 이런 UDF를 호출하는 예를 UDFcli.java와 UDFclie.sqlj 샘플에서 찾을 수 있습니다. 샘플 및 샘플의 컴파일과 수행에 대한 README 지침은 sqllib/samples/java 디렉토리에서 찾으십시오.

Java UDF 코딩

일반적으로 SQL 유형 *t1*, *t2* 및 *t3*를 받고, 유형 *t4*를 리턴하는 UDF를 선언하는 경우, 이것은 예상된 Java 시그니쳐를 사용하는 Java 메소드라고 부릅니다.

```
public void name ( T1 a, T2 b, T3 c, T4 d)  { .....}
```

여기에서,

- *name*은 메소드 이름입니다.
- *T1*에서 *T4*까지는 SQL 유형 *t1*에서 *t4*까지에 대응하는 Java 유형입니다.
- *a*, *b* 및 *c*는 입력 인수에 대한 임의의 변수 이름입니다.
- *d*는 계산되고 있는 UDF 결과를 나타내는 임의의 변수 이름입니다.

예를 들어, INTEGER를 리턴하고, CHAR(5), BLOB(10K) 및 DATE 유형의 인수를 받아들이는 sample!test3이라는 UDF가 있을 때, DB2에서 UDF의 Java 구현시 다음과 같은 시그니처가 있어야 합니다.

```
import COM.ibm.db2.app.*;
public class sample extends UDF {
   public void test3(String arg1, Blob arg2, String arg3,
                     int result) { ... }
}
```

테이블 함수를 구현하는 Java UDF에는 더 많은 인수가 필요합니다. 입력을 나타내는 변수들 이외에 추가적인 변수가 결과 행의 각 컬럼에 표시됩니다. 예를 들어 테이블 함수는 다음과 같이 선언될 수 있습니다.

```
public void test4(String arg1, int result1,
                  Blob result2, String  result3);
```

SQL NULL 값은 초기화되지 않은 Java 변수에 의해 표시됩니다. 이 변수들은 원시(primitive) 유형인 경우 0 값을 갖고, 오브젝트 유형인 경우, Java 규칙에 따라 Java 널(NULL) 값을 갖습니다. SQL NULL을 일반적인 0과 구분하려면, 입력 인수에 대하여 isNull 함수를 호출할 수 있습니다.

```
{ ....
   if (isNull(1)) { /* argument #1 was a SQL NULL */ }
   else           { /* not NULL */ }
}
```

위의 예에서 인수 번호는 1부터 시작합니다. isNull() 함수는 다음에 나오는 다른 함수들과 마찬가지로 COM.ibm.db2.app.UDF 클래스로부터 계승됩니다.

결과를 스칼라 또는 테이블 UDF로부터 리턴하려면, 다음과 같이 UDF에서 set() 메소드를 사용하십시오.

```
{ ....
   set(2, value);
}
```

여기에서 '2'는 출력 인수 색인이고, value는 호환가능한 유형의 리터럴 또는 변수입니다. 인수 번호는 선택된 출력의 인수 목록의 색인입니다. 이 절의 첫번째 예

에서 `int result` 변수는 색인 4를 갖습니다. 두번째로 `result1`에서 `result3`은 색인 2에서 4를 갖습니다. UDF가 리턴하기 전에 설정되지 않은 출력 인수는 널(NULL) 값을 갖게 됩니다.

UDF와 저장 프로시듀어에서 사용되는 C 모듈과 마찬가지로, Java UDF에서 Java 표준 I/O 스트림(`System.in`, `System.out`, 및 `System.err`)을 사용할 수 없습니다. Java UDF의 예는 `sqllib/samples/java` 디렉토리의 `DB2Udf.java` 파일을 참조하십시오.

UDF를 구현하기 위해 사용하는 모든 Java 클래스 파일들은 `sqllib/function directory` 또는 적절한 서브디렉토리에 들어 있어야 함을 명심하십시오. 741 페이지의 『Java 클래스를 넣을 위치』의 내용을 참조하십시오.

Java UDF 실행 방법 변경

일반적으로 DB2는 UDF를 여러번 호출하는데, 조회에서 입력 또는 결과 세트의 각 행에 한 번씩 호출하게 됩니다. SCRATCHPAD가 UDF의 CREATE FUNCTION문에 지정된 경우, DB2는 연속된 UDF 호출 사이에 특정 "연속성"이 필요함을 인식하므로, Java 클래스 구현이 각 호출에 대하여 인스턴스화되지 않으나, 일반적으로 명령문 하나의 UDF 참조 하나에 한 번 호출합니다. 일반적으로 첫번째 호출 이전에 인스턴스화되고 그 후에 사용되지만, 테이블 함수의 경우 더 자주 인스턴스화될 수 있습니다. 더 자세한 정보는 다음에 나오는 NO FINAL CALL 실행 모델을 참조하십시오.

그러나 NO SCRATCHPAD가 UDF 또는 스칼라나 테이블 함수 중 하나에 지정되면, 클린 인스턴스가 UDF에 대한 각각의 호출에 대해 인스턴스화됩니다.

스크래치 패드는 UDF에 대한 복수 호출에 걸쳐 정보를 저장할 때 유용합니다. Java와 OLE UDF가 호출들 간의 연속성을 유지하기 위해 인스턴스 변수를 사용하거나 스크래치 패드를 설정할 수 있는 반면, C나 C++는 스크래치 패드를 사용해야 합니다. Java UDF는 `COM.ibm.db2.app.UDF`에서 사용할 수 있는 getScratchPad()와 setScratchPad() 메소드를 사용하여 스크래치 패드에 액세스합니다.

스크래치 패드를 사용하는 Java 테이블 함수의 경우, 469 페이지의 『Java에 대한 테이블 함수 실행 모델』의 실행 모델에 의해 알 수 있듯이 CREATE FUNCTION

문에 FINAL CALL이나 NO FINAL CALL 옵션을 사용하여 새로운 스크래치 패드 인스턴스를 확보하는 시점을 제어하십시오.

스크래치 패드를 통해 UDF에 대한 호출들 간의 연속성을 유지할 수 있는 기능은, DB2 스크래치 패드나 인스턴스 변수 사용 여부에 상관없이 CREATE FUNCTION의 SCRATCHPAD와 NO SCRATCHPAD 옵션에 의해 제어됩니다.

스칼라 함수의 경우, 전체 명령문에 동일한 인스턴스를 사용합니다.

같은 UDF가 여러번 참조되더라도 조회에서 Java UDF에 대한 모든 참조는 독립적으로 처리된다는 점에 주의하십시오. 이것은 OLE, C 및 C++ UDF에서 발생하는 것과 같습니다. 스칼라 함수에 FINAL CALL 옵션을 지정한 경우, 조회 종료시 오브젝트의 close() 메소드가 호출됩니다. 테이블 함수의 경우, close() 메소드는 항상 다음에 나오는 내용과 같이 호출됩니다. UDF 클래스에 close() 메소드를 정의하지 않는 경우, 스텁 함수가 제어를 받아 이벤트가 무시됩니다.

CREATE FUNCTION문에 Java에 대한 ALLOW PARALLEL절을 지정하는 경우, DB2는 UDF의 병렬 평가를 선택할 수도 있습니다. 이러한 경우, 여러 개의 독립된 Java 오브젝트가 서로 다른 파티션에서 작성될 수 있습니다. 각 오브젝트는 행의 부속 집합을 수신합니다.

다른 UDF에서와 마찬가지로 Java UDF는 FENCED 또는 NOT FENCED 될 수 있습니다. NOT FENCED UDF는 데이터베이스 엔진의 주소 공간 내에서 수행됩니다. FENCED UDF는 별도의 프로세스에서 수행됩니다. Java UDF가 삽입되어 있는 프로세스의 주소 공간을 손상시킬 수는 없어도, 프로세스를 종료하거나 속도를 느리게 할 수 있습니다. 그러므로 Java로 작성된 UDF를 디버그할 때, 이들을 FENCED UDF로 수행해야 합니다.

`COM.ibm.db2.app.UDF`에 대한 설명은 867 페이지의 『COM.ibm.db2.app.UDF』의 내용을 참조하십시오. 이 인터페이스는 `setSQLstate` 및 `getDBinfo`와 같은 UDF 내에서 호출할 수 있는 다른 유용한 호출들을 설명합니다.

Java에 대한 테이블 함수 실행 모델

Java로 작성된 테이블 함수의 경우, 테이블 함수에 중요한 해당 명령문의 DB2 처리 상의 각 지점에서 어떠한 일들이 발생하는지를 이해하는 것이 중요합니다. 다

음의 테이블은 이 정보에 대한 자세한 설명입니다. 각 상자의 아랫 부분에서는 웹에서 일부 정보를 가져오는 일반적인 테이블 함수에 대하여 이 코드를 작성하여 할 수 있는 것이 무엇인지를 보여줍니다. NO FINAL CALL과 FINAL CALL의 두 가지 경우 모두 설명되며, 이 두 가지 경우 모두에 SCRATCHPAD가 있는 것으로 가정합니다.

스캔시 시점	**NO FINAL CALL LANGUAGE JAVA SCRATCHPAD**	**FINAL CALL LANGUAGE JAVA SCRATCHPAD**
테이블 함수의 첫번째 OPEN 전	호출 없음.	• 클래스 구성자가 호출됩니다(새로운 스크래치 패드를 의미). UDF 메소드는 FIRST 호출과 함께 호출됩니다. • 구성자는 클래스와 스크래치 패드 변수를 초기화합니다. 메소드는 웹 서버에 연결됩니다.
테이블 함수의 각 OPEN시	• 클래스 구성자가 호출됩니다(새로운 스크래치 패드를 의미). UDF 메소드는 OPEN 호출과 함께 호출됩니다. • 구성자는 클래스와 스크래치 패드 변수를 초기화합니다. 메소드는 웹 서버에 연결되고 웹 데이터에 대한 스캔을 엽니다.	• UDF 메소드가 OPEN 호출과 함께 열립니다. • 메소드는 원하는 웹 데이터에 대한 스캔을 엽니다. 스크래치 패드에 저장되어 있는 것에 따라, CLOSE 위치 재지정 후 다시 열기를 방지할 수도 있습니다.
테이블 함수 데이터의 새로운 행에 대한 각 FETCH시	• UDF 메소드가 FETCH 호출과 함께 호출됩니다. • 메소드는 데이터의 다음 행 또는 EOT를 페치하고 리턴합니다.	• UDF 메소드가 FETCH 호출과 함께 호출됩니다. • 메소드는 데이터의 새로운 행 또는 EOT를 페치하고 리턴합니다.
테이블 함수의 각 CLOSE시	• UDF 메소드는 CLOSE 호출과 함께 호출됩니다. 클래스에 대하여 존재하는 경우, `close()` 메소드. • 메소드는 웹 스캔을 닫고 웹 서버와의 연결을 해제합니다. `close()`는 아무 것도 할 필요가 없습니다.	• UDF 메소드는 CLOSE 호출과 함께 호출됩니다. • 메소드는 스캔의 맨 위로 위치를 재지정하거나 스캔을 닫을 수 있습니다. 이것은 스크래치 패드의 상태를 저장할 수 있는데, 이것은 지속됩니다.

스캔시 시점	NO FINAL CALL LANGUAGE JAVA SCRATCHPAD	FINAL CALL LANGUAGE JAVA SCRATCHPAD
테이블 함수의 최종 CLOSE 후	호출 없음.	• UDF 메소드는 FINAL 호출과 함께 호출됩니다. 클래스에 대하여 존재하는 경우, close() 메소드. • 메소드는 웹 서버와의 연결을 해제합니다. close()는 아무 것도 할 필요가 없습니다.

주:

1. "UDF 메소드"란 UDF를 구현하는 Java 클래스 메소드를 말합니다. 이것은 CREATE FUNCTION문의 EXTERNAL NAME절에서 식별되는 메소드입니다.
2. NO SCRATCHPAD가 지정된 테이블 함수의 경우, UDF 메소드에 대한 호출은 이 테이블에 지정된대로이지만, 사용자가 스크래치 패드를 통해 어떠한 연속성도 요구하지 않으므로 DB2는 클래스 구성자를 호출함으로써, 각 호출 전에 새로운 오브젝트가 인스턴스화 되도록 합니다. NO SCRATCHPAD가 지정된(그래서 연속성이 없는) 테이블 함수가 매우 유용한 작업을 수행할 수 있는지는 확실하지 않지만, 지원은 됩니다.
3. 이 모델들은 다른 UDF 언어들, 즉, C/C++ 및 OLE에서 수행되는 것들과 완벽하게 호환됩니다.

OLE 자동 UDF 작성

OLE(오브젝트 링크 및 삽입) 자동화는 Microsoft Corporation의 OLE 2.0 구조의 일부입니다. OLE 자동화를 사용하여, 사용자의 응용프로그램은 작성된 언어에 상관없이 OLE 자동화 오브젝트에서 특성과 메소드를 공개할 수 있습니다. Lotus Notes 또는 Microsoft Exchange®과 같은 기타 응용프로그램은 이런 특성과 메소드를 OLE 자동화를 통해 사용하여 이런 오브젝트를 통합할 수 있습니다.

특성과 메소드를 공개하는 응용프로그램들은 OLE 자동화 서버 또는 오브젝트라고 하며, 이 특성과 메소드에 액세스하는 응용프로그램들을 OLE 자동화 제어기라

고 합니다. OLE 자동화 서버는 OLE IDispatch 인터페이스를 구현하는 COM 구성요소(오브젝트)입니다. OLE 자동화 제어기는 IDispatch 인터페이스를 통해 자동화 서버와 통신하는 COM 클라이언트입니다. COM(구성요소 오브젝트 모델)은 OLE의 기초입니다. OLE 자동화 UDF의 경우, DB2는 OLE 자동화 제어기로 작동합니다. 이러한 메카니즘을 통해 DB2는 OLE 자동화 오브젝트 메소드를 외부 UDF로서 호출할 수 있습니다.

이 절에서는 사용자가 OLE 자동화 용어 및 개념에 익숙하다고 가정합니다. 이 책에서는 OLE에 대한 소개를 제공하지 않습니다. OLE 자동화 개요는 *Microsoft Corporation: The Component Object Model Specification*, October 1995를 참조하십시오. OLE 자동화에 대한 자세한 내용은 *OLE Automation Programmer's Reference*, Microsoft Press, 1996, ISBN 1-55615-851-3을 참조하십시오.

OLE 자동화 UDF를 보여주는 DB2 응용프로그램 개발 클라이언트와 함께 포함된 샘플 응용프로그램 목록은 854 페이지의 표49의 내용을 참조하십시오.

OLE 자동화 UDF 작성 및 등록

OLE 자동화 UDF는 OLE 자동화 오브젝트의 공용 메소드로 구현됩니다. OLE 자동화 오브젝트는 OLE 자동화 제어기, 즉 이 경우 DB2에 의해 외부에서 작성되어야 하고, 추후 바인딩(IDispatch 기초 바인딩)을 지원합니다. OLE 자동화 오브젝트는 클래스 식별자(CLSID) 또는 선택적으로 자동화 오브젝트를 식별하기 위해 OLE 프로그램 ID(progID)와 함께 Windows 등록 데이터베이스(레지스트리)에 등록되어야 합니다. progID는 DCOM(분산 COM)을 통해 내부 프로세스(.DLL)나 지역(.EXE) OLE 자동화 서버 또는 원격 서버를 식별할 수 있습니다. OLE 자동화 UDF는 스칼라 함수 또는 테이블 함수가 될 수 있습니다.

OLE 자동화 오브젝트를 코딩한 후, SQL CREATE FUNCTION문을 사용하여 오브젝트의 메소드를 UDF로 등록해야 합니다. OLE 자동화 UDF를 등록하는 것은 외부 C나 C++ UDF를 등록하는 것과 매우 비슷하지만, 다음의 옵션을 사용해야 합니다.

- LANGUAGE OLE
- FENCED(OLE 자동화 UDF는 FENCED 모드에서 수행되어야 함)

외부 이름은 OLE 자동화 오브젝트를 식별하는 OLE progID와 !(느낌표)로 구분되는 메소드 이름으로 구성됩니다.

```
CREATE FUNCTION bcounter () RETURNS INTEGER
   EXTERNAL NAME 'bert.bcounter!increment'
    LANGUAGE OLE
    FENCED
    SCRATCHPAD
    FINAL CALL
    NOT DETERMINISTIC
    NULL CALL
    PARAMETER STYLE DB2SQL
    NO SQL
NO EXTERNAL ACTION
    DISALLOW PARALLEL;
```

OLE 메소드 구현에 대한 호출 규약은 C나 C++로 작성된 함수에 대한 규약과 동일합니다. BASIC 언어로 된 위 메소드의 구현은 아래와 같습니다(BASIC에서 매개변수는 기본적으로 참조에 의한 호출로 정의됨에 주의).

```
Public Sub increment(output As Long, _
                    indicator As Integer, _
                     sqlstate As String, _
                     fname As String, _
                     fspecname As String, _
                    sqlmsg As String, _
                     scratchpad() As Byte, _
                     calltype As Long)
```

오브젝트 인스턴스 및 스크래치 패드 고려사항

OLE 자동화 UDF(OLE 자동화 오브젝트 메소드)는 OLE 자동화 오브젝트의 인스턴스에 적용됩니다. DB2는 SQL문에서 각 UDF 참조에 대한 오브젝트 인스턴스를 작성합니다. 오브젝트 인스턴스는 SQL문에서 UDF 참조의 후속 메소드 호출에 재사용되거나, 이 인스턴스는 메소드 호출 후 릴리스되고 새로운 인스턴스가 각 후속 메소드 호출에 작성됩니다. 적절한 방법은 SQL CREATE FUNCTION 문에 SCRATCHPAD 옵션과 함께 지정될 수 있습니다. LANGUAGE OLE절의 경우, SCRATCHPAD 옵션은 하나의 오브젝트 인스턴스가 전체 조회에 대하여 작성되고 재사용되는 C나 C++에 비해 추가적인 구문들이 있는데 반해, NO SCRATCHPAD가 지정되면, 새로운 오브젝트 인스턴스는 메소드가 호출될 때마다 작성됩니다. SQL문에서 각 UDF 참조에 대하여 별도의 인스턴스들이 작성됩니다.

스크래치 패드를 사용함으로써 메소드가 여러 함수 호출에 걸쳐 오브젝트의 인스턴스 변수에 상태 정보를 유지보수할 수 있습니다. 또한 오브젝트 인스턴스가 단 한 번 작성되고 후속 호출에 재사용되므로 성능을 향상시킵니다.

SQL 데이터 유형이 OLE 자동화 UDF로 전달되는 방식

DB2는 SQL 유형과 OLE 자동화 유형간의 유형변환을 처리합니다. 다음의 테이블에서는 지원되는 데이터 유형 및 이들이 맵핑되는 방식을 요약했습니다. OLE 자동화 유형을 BASIC이나 C/C++와 같은 구현하는 프로그래밍 언어의 데이터 유형으로 맵핑하는 것에 대한 내용은 475 페이지의 표17에 설명되어 있습니다.

표 16. SQL 및 OLE 자동화 데이터 유형의 맵핑

<table>
<tr><th>SQL 유형</th><th>OLE 자동 유형</th><th>OLE 자동 유형 설명</th></tr>
<tr><td>SMALLINT</td><td>short</td><td>16 비트 부호있는 정수</td></tr>
<tr><td>INTEGER</td><td>long</td><td>32 비트 부호있는 정수</td></tr>
<tr><td>REAL</td><td>float</td><td>32 비트 IEEE 유동소수점 숫자</td></tr>
<tr><td>FLOAT 또는 DOUBLE</td><td>double</td><td>64 비트 IEEE 유동소수점 숫자</td></tr>
<tr><td>DATE</td><td>DATE</td><td rowspan="3">1899년 12월 30일 이후 날수의 64 비트 유동소수점 나머지 수</td></tr>
<tr><td>TIME</td><td>DATE</td></tr>
<tr><td>TIMESTAMP</td><td>DATE</td></tr>
<tr><td>CHAR(n)</td><td>BSTR</td><td rowspan="4">OLE 자동화 프로그래머 참조서에서 설명된 길이가 접두어로 붙는 문자열.</td></tr>
<tr><td>VARCHAR(n)</td><td>BSTR</td></tr>
<tr><td>LONG VARCHAR</td><td>BSTR</td></tr>
<tr><td>CLOB(n)</td><td>BSTR</td></tr>
<tr><td>GRAPHIC(n)</td><td>BSTR</td><td rowspan="4">OLE 자동화 프로그래머 참조서에서 설명된 길이가 접두어로 붙는 문자열.</td></tr>
<tr><td>VARGRAPHIC(n)</td><td>BSTR</td></tr>
<tr><td>LONG GRAPHIC</td><td>BSTR</td></tr>
<tr><td>DBCLOB(n)</td><td>BSTR</td></tr>
<tr><td>CHAR(n)[1]</td><td>SAFEARRAY[부호없는 char]</td><td rowspan="4">8 비트 부호없는 데이터 항목의 1차원 Byte() 배열. SAFEARRAY는 OLE 자동화 프로그래머 참조서에 설명되어 있습니다.</td></tr>
<tr><td>VARCHAR(n)[1]</td><td>SAFEARRAY[부호없는 char]</td></tr>
<tr><td>LONG VARCHAR[1]</td><td>SAFEARRAY[부호없는 char]</td></tr>
<tr><td>BLOB(n)</td><td>SAFEARRAY[부호없는 char]</td></tr>
</table>

주:

1. FOR BIT DATA가 지정됨

DB2와 OLE 자동화 UDF 간에 전달된 데이터는 참조에 의한 호출로 전달됩니다. 테이블에 나열되지 않은 BIGINT, DECIMAL이나 LOCATORS와 같은 SQL 유형 또는 부울이나 CURRENCY와 같은 OLE 자동화 유형은 지원되지 않습니다. BSTR에 맵핑된 문자 또는 그래픽 데이터는 데이터베이스 코드 페이지에서 UCS-2(유니코드라고도 하는 IBM 코드 페이지 13488) 스킴으로 맵핑됩니다. 리턴시 데이터는 다시 데이터베이스 코드 페이지로 변환됩니다. 이러한 변환은 데이터베이스 코드 페이지에 상관없이 발생합니다. 데이터베이스 코드 페이지에서 UCS-2로 또는 UCS-2에서 데이터베이스 코드 페이지로 변환할 코드 페이지 변환 테이블이 설치되어 있지 않은 경우, SQLCODE -332(SQLSTATE 57017)를 받게 됩니다.

BASIC 및 C++에서 OLE 자동화 구현

모든 언어로 OLE 자동화 UDF를 구현할 수 있습니다. 이 절에서는 두 가지 샘플 언어로 BASIC과 C++를 사용하여 OLE 자동화 UDF를 구현하는 방법을 보여줍니다.

표17에서는 다양한 SQL 데이터 유형이 중간 생성되는 OLE 자동화 데이터 유형 및 해당 언어(BASIC 또는 C++)의 데이터 유형으로 맵핑되는 것을 보여줍니다. OLE 데이터 유형은 언어에 독립적입니다. 즉, 474 페이지의 표16은 모든 언어에 적용됩니다.

표 17. SQL 및 OLE 데이터 유형을 BASIC과 C++ 데이터 유형으로 맵핑

SQL 유형	**OLE 자동 유형**	**UDF 언어**	
		BASIC 유형	**C++ 유형**
SMALLINT	short	Integer	short
INTEGER	long	Long	long
REAL	float	Single	float
FLOAT 또는 DOUBLE	double	Double	double
DATE, TIME, TIMESTAMP	DATE	Date	DATE
CHAR(*n*), VARCHAR(*n*), LONG VARCHAR, CLOB(*n*)	BSTR	String	BSTR
GRAPHIC(*n*), VARGRAPHIC(*n*), LONG GRAPHIC, DBCLOB(*n*)	BSTR	String	BSTR
CHAR(*n*)[1], VARCHAR(*n*)[1], LONG VARCHAR[1], BLOB(*n*)	SAFEARRAY[부호없는 char]	Byte()	SAFEARRAY

표 17. SQL 및 OLE 데이터 유형을 BASIC과 C++ 데이터 유형으로 맵핑 (계속)

SQL 유형	OLE 자동 유형	UDF 언어	
		BASIC 유형	C++ 유형

주:

1. FOR BIT DATA가 지정됨

BASIC의 OLE 자동 UDF

BASIC으로 OLE 자동화 UDF를 구현하기 위해서는, OLE 자동화 유형에 맵핑되는 SQL 데이터 유형에 대응하는 BASIC 데이터 유형을 사용해야 합니다.

472 페이지의 『OLE 자동화 UDF 작성 및 등록』의 bcounter OLE 자동화 UDF의 BASIC 선언은 다음과 같습니다.

```
Public Sub increment(output As Long, _
                    indicator As Integer, _
                     sqlstate As String, _
                     fname As String, _
                     fspecname As String, _
                    sqlmsg As String, _
                     scratchpad() As Byte, _
                     calltype As Long)
```

OLE 테이블 자동화의 예는 532 페이지의 『예: BASIC으로 작성한 메일 OLE 자동 테이블 함수』에서 설명됩니다.

C++의 OLE 자동 UDF

475 페이지의 표17에서는 SQL 데이터 유형에 대응하는 C++ 데이터 유형 및 이들이 OLE 자동화 유형에 맵핑되는 방식을 보여줍니다.

increment OLE 자동화 UDF의 C++ 선언은 다음과 같습니다.

```
STDMETHODIMP Ccounter::increment (long   *output,
                                  short  *indicator,
                                 BSTR    *sqlstate,
                                 BSTR    *fname,
                                 BSTR    *fspecname,
                                  BSTR   *sqlmsg,
                                  SAFEARRAY **scratchpad,
                                         long *calltype );
```

OLE는 OLE 자동화 오브젝트의 특성과 메소드를 설명하는 유형 라이브러리를 지원합니다. 공개된 오브젝트, 특성 및 메소드는 오브젝트 설명 언어(ODL)에서 설명됩니다. 위의 C++ 메소드에 대한 ODL 설명은 다음과 같습니다.

```
HRESULT increment ([out]    long  *output,
                   [out]    short *indicator,
                      [out] BSTR    *sqlstate,
                      [in]  BSTR    *fname,
                      [in]  BSTR    *fspecname,
                   [out]    BSTR  *sqlmsg,
                   [in,out] SAFEARRAY (unsigned char) *scratchpad,
                      [in]  long    *calltype);
```

ODL 설명으로 매개변수가 입력(in) 매개변수인지, 출력(out) 매개변수인지 또는 입출력(in,out) 매개변수인지 여부를 지정할 수 있습니다. OLE 자동화 UDF의 경우, UDF 입력 매개변수와 입력 표시기는 [in] 매개변수로 지정되고, UDF 출력 매개변수와 출력 표시기는 [out] 매개변수로 지정됩니다. UDF 후미(trailing) 인수의 경우, sqlstate는 [out] 매개변수이고, 함수 이름 및 함수 고유 이름은 [in] 매개변수이며, 스크래치 패드는 [in,out] 매개변수이고, 호출 유형은 [in] 매개변수입니다.

테이블 함수에는 CREATE FUNCTION문의 RETURN 컬럼에 해당되는 여러개의 출력 매개변수와 출력 표시기가 있는데 비해, 스칼라 함수의 경우 하나의 출력 매개변수와 출력 표시기가 있습니다.

OLE 자동화는 문자열을 처리하기 위해 BSTR 데이터 유형을 정의합니다. BSTR은 OLECHAR에 대한 포인터(typedef OLECHAR *BSTR)로 정의됩니다. BSTR 할당 및 해제를 위해, OLE는 매개변수에 새로운 값을 지정하기 전에 참조에 의한 매개변수에 전달된 BSTR을 호출자가 해제하도록 규칙을 정했습니다. 이 규칙은 DB2와 OLE 자동화 UDF에 대하여 다음을 의미합니다. 동일한 규칙이 다음 SAFEARRAY**로 호출자에 의해 수신된 1차원 바이트 배열에 적용됩니다.

- [in] 매개변수: DB2는 [in] 매개변수를 할당하고 해제합니다.
- [out] 매개변수: DB2는 포인터를 널(NULL)로 전달합니다. [out] 매개변수는 호출자에 의해 할당되고 DB2에 의해 해제되어야 합니다.
- [in,out] 매개변수: DB2는 초기에 [in,out] 매개변수를 할당합니다. 이들은 호출자에 의해 해제되고 다시 할당될 수 있습니다. [out] 매개변수에서처럼 DB2는 마지막으로 리턴된 매개변수를 해제합니다.

다른 매개변수들은 포인터로 전달됩니다. DB2는 참조된 메모리를 할당하고 관리합니다.

OLE 자동화에서는 BSTR과 SAFEARRAY를 처리하기 위한 데이터 조작 함수 세트를 제공합니다. 데이터 조작 함수는 *OLE 자동화 프로그래머 참조서*에 설명되어 있습니다.

다음의 C++ UDF는 CLOB 입력 매개변수의 처음 5자를 리턴합니다.

```
// UDF DDL: CREATE FUNCTION crunch (clob(5k)) RETURNS char(5)

STDMETHODIMP Cobj::crunch (BSTR *in,          // CLOB(5K)
                           BSTR *out,         // CHAR(5)
                           short *indicator1,  // input indicator
                           short *indicator2,  // output indicator
                           BSTR *sqlstate,    // pointer to NULL
                           BSTR *fname,       // pointer to function name
                           BSTR *fspecname,   // pointer to specific name
                           BSTR *msgtext)     // pointer to NULL
  {
     // Allocate BSTR of 5 characters
     // and copy 5 characters of input parameter

     // out is an [out] parameter of type BSTR, that is,
     // it is a pointer to NULL and the memory does not have to be freed.
     // DB2 will free the allocated BSTR.

     *out = SysAllocStringLen (*in, 5);
  return NOERROR;
  };
```

OLE 자동 서버는 *작성가능한 단일 사용* 또는 *작성가능한 복수 사용*으로 구현될 수 있습니다. 작성가능한 단일 사용을 통해 CoGetClassObject를 사용하여 OLE 자동 오브젝트에 연결된 각 클라이언트(즉, DB2 FENCED 프로세스)는 클래스 팩토리(factory)의 자신의 인스턴스를 사용하고 필요에 따라 OLE 자동 서버의 새로운 사본을 수행시킬 수 있습니다. 작성가능한 복수 사용을 통해 많은 클라이언트들이 같은 클래스 팩토리(factory)로 연결됩니다. 즉, 클래스 팩토리(factory)의 각 인스턴스화는 이미 수행되고 있는 OLE 서버 사본이 있는 경우 이것에 의해 제공됩니다. 수행 중인 OLE 서버 사본이 없는 경우, 클래스 오브젝트를 제공하기 위해 사본이 자동으로 시작됩니다. 단일 사용 OLE 자동 서버와 복수 사용 OLE 자동 서버 중 무엇을 선택할 것인가는 자동 서버를 구현할 때 사용자가 선택하게 됩니다. 성능 상의 관점에서는 단일 사용 서버가 권장됩니다.

OLE DB 테이블 함수

Microsoft OLE DB는 응용프로그램에 다양한 정보 소스에 저장되어 있는 데이터에 일관된 액세스를 제공하는 OLE/COM 인터페이스 세트입니다. OLE DB 구성요소 DBMS 구조는 OLE DB 사용자와 OLE DB 제공자를 정의합니다. OLE DB 사용자는 OLE DB 인터페이스를 사용하는 시스템내 응용프로그램입니다. OLE DB 제공자는 OLE DB 인터페이스를 공개하는 구성요소입니다. OLE DB 제공자에는 두 가지 클래스가 있습니다. 데이터를 소유하고 테이블 형식의 데이터를 행 집합으로 공개하는 *OLE DB 데이터 제공자*와, 자신의 데이터를 가지고 있지는 않으나 OLE DB 인터페이스를 통해 데이터를 만들고 사용함으로써 특정 서비스를 캡슐화하는 *OLE DB 서비스 제공자*입니다.

DB2 Universal Database는 사용자가 OLE DB 데이터 소스에 액세스하는 테이블 함수를 정의할 수 있도록 하여 OLE DB 응용프로그램 작성을 단순화합니다. DB2는 어떠한 OLE DB 데이터나 서비스 제공자에게도 액세스할 수 있는 OLE DB 사용자입니다. OLE DB 인터페이스를 통해 자신의 데이터를 공개하는 데이터 소스에 대하여 GROUP BY, JOIN 및 UNION 등의 조작을 수행할 수 있습니다. 예를 들어, Microsoft Access 데이터베이스나 Microsoft Exchange 주소록으로부터 테이블을 리턴하는 OLE DB 테이블 함수를 정의한 후, 이 OLE DB 테이블 함수로부터의 데이터를 사용자의 DB2 데이터베이스의 데이터와 경계없이 (seamlessly) 조합한 보고서를 작성할 수 있습니다.

OLE DB 테이블 함수를 사용하면 임의의 OLE DB 제공자로의 내장 액세스를 제공함으로써, 응용프로그램 개발 노력을 절감할 수 있습니다. C, Java 및 OLE 자동 테이블 함수의 경우 개발자가 테이블 함수를 구현해야 하는 반면, OLE DB 테이블 함수의 경우 원래의 내장 OLE DB 사용자 인터페이스가 OLE DB 제공자와 인터페이스하여 데이터를 검색합니다. 단지 언어 유형 OLEDB의 테이블 함수를 등록하고, 데이터 소스로서 OLE DB 제공자와 적절한 행 집합을 참조하기만 하면 됩니다. OLE DB 테이블 함수를 활용하기 위해 UDF 프로그래밍을 할 필요가 없습니다.

DB2 Universal Database와 함께 OLE DB 테이블 함수를 사용하려면, OLE DB 2.0 이상을 설치해야 하는데, 이것은 `http://www.microsoft.com`의 Microsoft에서 구할 수 있습니다. 먼저 OLE DB를 설치하지 않고 OLE DB 테이블 함수

를 호출하려한 경우, DB2는 SQLCODE 465, SQLSTATE 58032,이유 코드 35를 발행합니다. 시스템 요구사항 및 사용자의 데이터 소스에서 사용할 수 있는 OLE DB 제공자에 대한 정보는 사용자의 데이터 소스 관련 문서를 참조하십시오. OLE DB 테이블 함수를 정의하고 사용하는 샘플 목록은 835 페이지의 『부록B. 샘플 프로그램』의 내용을 참조하십시오. OLE DB 스펙은 *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998을 참조하십시오.

OLE DB 테이블 함수 작성

하나의 CREATE FUNCTION문을 사용하여 OLE DB 테이블 함수를 정의하려면, 다음과 같이 하십시오.

- OLE DB 제공자가 리턴하는 테이블을 정의하십시오.
- LANGUAGE OLEDB를 지정하십시오.
- OLE DB 행 집합을 식별하고 EXTERNAL NAME절에 OLE DB 제공자 연결 문자열을 지정하십시오.

OLE DB 데이터 소스는 자신의 데이터를 *행 집합*이라는 테이블 형식으로 공개합니다. 행 집합은 각각의 행이 컬럼 세트로 구성되는 행의 세트입니다. RETURNS TABLE절에는 사용자와 관련있는 컬럼만이 포함됩니다. 테이블 함수 컬럼의 OLE DB 데이터 소스의 행 집합 컬럼으로의 바인딩은 컬럼 이름에 근거합니다. OLE DB 제공자가 대소문자를 구별하는 경우, 컬럼 이름을 인용부호 안에 넣으십시오. 예를 들면 "UPPERcase"입니다. 완전한 이름이 될 수 있는 행 집합 이름에 대한 정보는 482 페이지의 『완전한 행 집합 이름』의 내용을 참조하십시오. OLE DB 데이터 유형의 DB2 데이터 유형으로의 맵핑에 대한 정보는 485 페이지의 『지원되는 OLE DB 데이터 유형』의 내용을 참조하십시오. CREATE FUNCTION문의 완전한 구문과 EXTERNAL NAME절에 대한 규칙은 *SQL 참조서*에서 자세한 내용을 참조하십시오.

EXTERNAL NAME절은 다음과 같은 형식 중 하나를 사용할 수 있습니다.

```
   'server!rowset'
또는
   '!rowset!connectstring'
```

여기서,

server CREATE SERVER문을 사용하여 등록된 서버를 식별합니다.

rowset

OLE DB 제공자에 의해 공개된 행 이름이나 테이블을 식별합니다. 테이블에 입력 매개변수가 있는 경우, OLE DB 제공자로 명령 텍스트를 전달하기 위해 이 값은 공백이어야 합니다.

connectstring

OLE DB 제공자에 연결하는 데 필요한 초기화 특성이 들어 있습니다. 연결 문자열의 완전한 구문 및 문법에 대한 정보는 *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998의 ″Data Link API of the OLE DB Core Components″를 참조하십시오.

CREATE FUNCTION문의 EXTERNAL NAME절에 *연결 문자*열을 사용하거나 CREATE SERVER문에 *CONNECTSTRING* 옵션을 지정할 수 있습니다.

예를 들어, 다음과 같은 CREATE FUNCTION과 SELECT문을 사용하여 OLE DB 테이블 함수를 정의하고 Microsoft Access 데이터베이스로부터의 테이블을 리턴할 수 있습니다.

```
CREATE FUNCTION orders ()
  RETURNS TABLE (orderid INTEGER, ...)
  LANGUAGE OLEDB
  EXTERNAL NAME '!orders!Provider=Microsoft.Jet.OLEDB.3.51;
                Data Source=c:\msdasdk\bin\oledb\nwind.mdb';

SELECT orderid, DATE(orderdate) AS orderdate,
                DATE(shippeddate) AS shippeddate
FROM TABLE(orders()) AS t
WHERE orderid = 10248;
```

EXTERNAL NAME절에 연결 문자열을 넣는 대신 서버 이름을 작성하고 사용할 수 있습니다. 예를 들어, 484 페이지의 『OLE DB 제공자의 서버 이름 정의』에서 설명된대로 서버 Nwind를 정의했다고 가정하면, 다음과 같은 CREATE FUNCTION문을 사용할 수 있습니다.

```
CREATE FUNCTION orders ()
  RETURNS TABLE (orderid INTEGER, ...)
  LANGUAGE OLEDB
  EXTERNAL NAME 'Nwind!orders';
```

OLE DB 테이블 함수를 사용하여 사용자는 문자열 데이터 유형의 입력 매개변수를 하나 지정할 수 있습니다. 입력 매개변수를 사용하여 명령 텍스트를 직접 OLE DB 제공자로 전달할 수 있습니다. 입력 매개변수를 정의한 경우, EXTERNAL NAME절에 행 집합 이름을 지정하지 마십시오. DB2는 실행을 위해 명령 텍스트를 OLE DB 제공자에게 전달하고, OLE DB 제공자는 행 집합을 DB2로 리턴합니다. 결과 행 집합의 컬럼 이름과 데이터 유형은 CREATE FUNCTION문의 RETURNS TABLE 정의와 호환가능해야 합니다. 행 집합의 컬럼 이름에 대한 바인딩은 일치하는 컬럼 이름에 근거하므로, 컬럼 이름을 적절하게 지정해야 합니다.

다음의 예에서는 OLE DB 테이블 함수를 등록하는데, 이 함수는 Microsoft SQL Server 7.0™ 데이터베이스에 저장된 정보를 검색합니다. 연결 문자열은 EXTERNAL NAME절에 지정됩니다. 테이블 함수에 OLE DB 제공자로 명령 텍스트를 통해 전달할 입력 매개변수가 있으므로, 행 집합 이름은 EXTERNAL NAME절에 지정되지 않습니다. 조회 예에서는 SQL 서버 데이터베이스로부터 세 개의 최상위 정보를 검색하는 SQL 명령 텍스트를 전달합니다.

```
CREATE FUNCTION favorites (varchar(600))
  RETURNS TABLE (store_id char (4), name varchar (41), sales integer)
  SPECIFIC favorites
  LANGUAGE OLEDB
  EXTERNAL NAME '!!Provider=SQLOLEDB.1;Persist Security Info=False;
  User ID=sa;Initial Catalog=pubs;Data Source=WALTZ;
  Locale Identifier=1033;Use Procedure for Prepare=1;
  Auto Translate=False;Packet Size=4096;Workstation ID=WALTZ;
  OLE DB Services=CLIENTCURSOR;';

SELECT *
FROM TABLE (favorites (' select top 3 sales.stor_id as store_id,  ' ||
                       '        stores.stor_name as name,          ' ||
                       '        sum(sales. qty) as sales           ' ||
                       ' from sales, stores                        ' ||
                       ' where sales.stor_id = stores.stor_id      ' ||
                       ' group by sales.stor_id, stores.stor_name ' ||
                       ' order by sum(sales.qty) desc')) as f;
```

완전한 행 집합 이름

일부 행 집합은 *완전한 이름*을 사용하여 EXTERNAL NAME절에서 식별될 수 있습니다. 완전한 이름은 다음 중 하나 또는 모두를 조합할 수 있습니다.

- 다음과 같은 정보를 필요로 하는 연관된 카탈로그 이름.
  - 제공자가 카탈로그 이름을 지원하는지 여부
  - 완전한 이름에서 카탈로그 이름을 넣을 위치

– 사용할 카탈로그 이름 구분 문자

- 다음과 같은 정보를 필요로 하는 연관된 스키마 이름.
  - 제공자가 스키마 이름을 지원하는지 여부
  - 사용할 스키마 이름 구분 문자

카탈로그와 스키마 이름에 대하여 사용자의 OLE DB 제공자가 제공하는 지원에 대한 정보는, 사용자의 OLE DB 제공자의 리터럴 정보 관련 문서를 참조하십시오.

DBLITERAL_CATALOG_NAME이 사용자의 제공자의 리터럴 정보에서 NULL이 아닌 경우, 카탈로그 이름 및 구분 문자로 DBLITERAL_CATALOG_SEPARATOR 값을 사용하십시오. 카탈로그 이름이 완전한 이름의 앞에 올 것인지 뒤에 올 것인지를 결정하려면, 사용자의 OLE DB 제공자의 특성 세트 DBPROPSET_DATASOURCEINFO에서 DBPROP_CATALOGLOCATION 값을 참조하십시오.

사용자의 제공자의 리터럴 정보에서 DBLITERAL_SCHEMA_NAME이 NULL이 아닌 경우, 스키마 이름과 구분 문자로 DBLITERAL_SCHEMA_SEPARATOR 값을 사용하십시오.

이름에 특수 문자가 들어 있거나 키워드와 동일한 경우, 이 이름을 사용자의 OLE DB 제공자에서 지정하는 인용 문자 안에 넣으십시오. 인용 문자는 사용자의 OLE DB 제공자 리터럴 정보에 DBLITERAL_QUOTE_PREFIX 및 DBLITERAL_QUOTE_SUFFIX로 정의되어 있습니다. 예를 들어, 다음의 EXTERNAL NAME에서 지정된 행 집합에는 *authors*라는 행 집합에 대한 카탈로그 이름 *pubs*와 스키마 이름 *dbo*가, 이름을 넣는 데 사용되는 인용 문자 "와 함께 포함됩니다.

```
EXTERNAL NAME '!"pubs"."dbo"."authors"!Provider=SQLOLEDB.1;...';
```

완전한 이름을 만드는 것에 대한 자세한 정보는 *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998 및 사용자의 OLE DB 제공자 관련 문서를 참조하십시오.

OLE DB 제공자의 서버 이름 정의

OLE 제공자에 대해 서버 이름을 정의하기 전에 CREATE WRAPPER OLEDB 문을 사용하여 데이터베이스마다 OLE DB 랩퍼를 한번 등록해야 합니다. CREATE WRAPPER OLEDB에 대한 자세한 내용은 *설치 및 구성 보충 설명서*를 참조하십시오.

많은 CREATE FUNCTION문에 대하여 사용자가 사용할 수 있는 OLE DB 데이터 소스의 서버 이름을 제공하려면, 다음과 같이 CREATE SERVER문을 사용하십시오.

- DB2 내에서 OLE DB 제공자를 식별하는 이름을 지정하십시오.
- WRAPPER OLEDB를 지정하십시오.
- CONNECTSTRING 옵션에 연결 정보를 지정하십시오.

예를 들어, 다음과 같은 CREATE SERVER문을 사용하여 Microsoft Access OLE DB 제공자에 대한 서버 이름 Nwind를 정의할 수 있습니다.

```
CREATE SERVER Nwind
  WRAPPER OLEDB
  OPTIONS (CONNECTSTRING 'Provider=Microsoft.Jet.OLEDB.3.51;
           Data Source=c:\msdasdk\bin\oledb\nwind.mdb');
```

그리고 서버 이름 Nwind를 사용하여 CREATE FUNCTION문에서 OLE DB 제공자를 식별할 수 있습니다. 예를 들면 다음과 같습니다.

```
CREATE FUNCTION orders ()
  RETURNS TABLE (orderid INTEGER, ...)
  LANGUAGE OLEDB
  EXTERNAL NAME 'Nwind!orders';
```

CREATE SERVER문의 완전한 구문에 대한 정보는 *SQL 참조서*에서 자세한 내용을 참조하십시오. OLE DB 제공자에 대한 사용자 맵핑 정보는 『사용자 맵핑 정의』의 내용을 참조하십시오.

사용자 맵핑 정의

DB2 사용자들에게 *사용자 맵핑*을 제공하여, 이들이 다른 사용자 이름과 암호를 가지고 OLE DB 데이터 소스에 액세스할 수 있도록 할 수 있습니다. 특정 사용자들의 사용자 이름을 맵핑하려면, CREATE USER MAPPING문을 사용하여 사

용자 맵핑을 정의할 수 있습니다. 모든 사용자들이 공유하는 사용자 맵핑을 제공하려면, 사용자 이름과 암호를 사용자의 CREATE FUNCTION 또는 CREATE SERVER문의 연결 문자열에 추가하십시오. 예를 들어, OLE DB 서버 Nwind에서 DB2 사용자 JOHN에 대한 특정 사용자 맵핑을 작성하려면, 다음과 같은 CREATE USER MAPPING문을 사용하십시오.

```
CREATE USER MAPPING FOR john
  SERVER Nwind
  OPTIONS (REMOTE_AUTHID 'dave', REMOTE_PASSWORD 'mypwd');
```

OLE DB 테이블 함수 orders를 호출하는 모든 DB2 사용자에게 동등한 액세스를 제공하려면, CREATE FUNCTION이나 CREATE SERVER문 중 하나에서 다음과 같은 CONNECTSTRING을 사용하십시오.

```
CREATE FUNCTION orders ()
  RETURNS TABLE (orderid INTEGER, ...)
  LANGUAGE OLEDB
  EXTERNAL NAME '!orders!Provider=Microsoft.Jet.OLEDB.3.51;User ID=dave;
                Password=mypwd;Data Source=c:\msdasdk\bin\oledb\nwind.mdb';
```

DB2는 다음과 같은 사용자 맵핑 규칙을 적용합니다.

- 사용자 맵핑이 정의되면, DB2는 맵핑된 권한 부여 정보를 사용하여 OLE DB 제공자에 연결하는데, 잠재적으로 CONNECTSTRING의 기존 사용자 ID와 암호를 겹쳐 씁니다.
- 사용자 맵핑이 정의되지 않으면, DB2는 권한 부여 정보가 제공되는 경우, CONNECTSTRING으로부터 권한 부여 정보를 사용합니다.
- 사용자 맵핑이 정의되지 않았고 CONNECTSTRING에 권한 부여 정보도 제공되지 않은 경우, DB2는 제공자가 권한 부여를 지원하는 경우, 현재의 DB2 권한 부여 정보를 사용합니다.

CREATE USER MAPPING문의 완전한 구문에 대한 정보는 *SQL 참조서*에서 자세한 내용을 참조하십시오.

지원되는 OLE DB 데이터 유형

연결 문자열의 완전한 구문 및 문법에 대한 정보는 *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998의 ″Data Link API of the OLE DB Core Components″를 참조하십시오. 사용자의 OLE DB 테이블 함수에서 적절한 RETURNS TABLE 컬럼을 정의하려면, 맵핑 테이

블을 사용하십시오. 예를 들어, 데이터 유형이 INTEGER인 컬럼을 갖는 OLE DB 테이블 함수를 정의하는 경우, DB2는 OLE DB 제공자로부터의 데이터를 DBTYPE_I4로서 요청합니다.

OLE DB 제공자 소스 데이터 유형을 OLE DB 데이터 유형에 맵핑하는 것에 대한 정보는 OLE DB 제공자 관련 문서를 참조하십시오. ANSI SQL, Microsoft Acess 및 Microsoft SQL 서버 제공자가 각각의 데이터 유형을 OLE DB 데이터 유형에 맵핑하는 방식의 예는 *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998을 참조하십시오.

표 18. DB2 데이터 유형을 OLE DB에 맵핑

DB2 데이터 유형	OLE DB 데이터 유형
SMALLINT	DBTYPE_I2
INTEGER	DBTYPE_I4
BIGINT	DBTYPE_I8
REAL	DBTYPE_R4
FLOAT/DOUBLE	DBTYPE_R8
DEC (p, s)	DBTYPE_NUMERIC (p, s)
DATE	DBTYPE_DBDATE
TIME	DBTYPE_DBTIME
TIMESTAMP	DBTYPE_DBTIMESTAMP
CHAR(N)	DBTYPE_STR
VARCHAR(N)	DBTYPE_STR
LONG VARCHAR	DBTYPE_STR
CLOB(N)	DBTYPE_STR
CHAR(N) FOR BIT DATA	DBTYPE_BYTES
VARCHAR(N) FOR BIT DATA	DBTYPE_BYTES
LONG VARCHAR FOR BIT DATA	DBTYPE_BYTES
BLOB(N)	DBTYPE_BYTES
GRAPHIC(N)	DBTYPE_WSTR
VARGRAPHIC(N)	DBTYPE_WSTR
LONG GRAPHIC	DBTYPE_WSTR
DBCLOB(N)	DBTYPE_WSTR

주: OLE DB 데이터 유형변환 규칙은 *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998에 정의되어 있습니다. 예를 들면 다음과 같습니다.

- OLE DB 데이터 유형 DBTYPE_CY를 검색하기 위해서, 데이터는 DB2 데이터 유형 DEC(19,4)로 맵핑되는 OLE DB 데이터 유형 DBTYPE_NUMERIC(19,4)로 변환될 수 있습니다.
- OLE DB 데이터 유형 DBTYPE_I1을 검색하기 위해서, 데이터는 DB2 데이터 유형 SMALLINT로 맵핑되는 OLE DB 데이터 유형 DBTYPE_I2로 변환될 수 있습니다.

- OLE DB 데이터 유형 DBTYPE_GUID를 검색하기 위해서, 데이터는 DB2 데이터 유형 CHAR(12) FOR BIT DATA로 맵핑되는 OLE DB 데이터 유형 DBTYPE_BYTES로 변환될 수 있습니다.

스크래치 패드 고려사항

사용자의 UDF에서 스크래치 패드를 사용할 것인지 여부에 영향을 미치는 인수는 이러한 특별한 절에서 설명해야 할만큼 중요합니다. 기타 코딩시 고려사항은 499 페이지의 『기타 코딩시 고려사항』에서 논의됩니다.

UDF를 재입력되도록 코딩하는 것이 중요합니다. 이것은 주로 UDF에 대한 많은 참조들이 함수 내용의 동일한 사본을 사용한다는 사실 때문입니다. 사실상 이러한 많은 참조들은 다른 명령문이나 응용프로그램에 있을 수도 있습니다. 그러나 함수들이 한 호출에서 다음 호출로의 상태를 저장할 필요가 있거나 저장하고자 한다는 점에 주의하십시오. 이러한 함수들의 두 가지 범주는 다음과 같습니다.

1. 저장 상태에 의존하는 함수들

   이러한 함수의 예는 맨 처음 호출되었을 때 '1'을 리턴하고, 그 이후 연속되는 호출에서 1씩 증가시키는 단순한 *계수기(counter)* 함수입니다. 이러한 함수는 SELECT 결과 행에 번호를 붙이는 데 사용할 수 있습니다.

   ```
   SELECT counter(), a, b+c, ...
   FROM tablex
   WHERE ...
   ```

   이러한 함수의 유형은 NOT DETERMINISTIC 또는 VARIANT입니다. 이 함수의 출력은 전적으로 SQL 인수 값에만 의존하지 않습니다. 이 *계수기 (counter)* 함수는 515 페이지의 『예: 계수기(Counter)』에 나와 있습니다.

2. 단 한번 초기화 조치를 수행하는 기능으로 성능이 향상될 수 있는 함수.

   이러한 함수의 예로, 문서 응용프로그램의 일부가 될 수도 있는 *일치(match)* 함수가 있는데, 이것은 주어진 문서에 주어진 문자열이 들어 있으면 'Y'를 리턴하고, 그렇지 않으면 'N'을 리턴합니다.

   ```
   SELECT docid, doctitle, docauthor
   FROM docs
   WHERE match('myocardial infarction', docid) = 'Y'
   ```

이 명령문은 첫번째 인수로 전달된 특정 텍스트 문자열 값이 들어 있는 모든 문서를 리턴합니다. 수행되는 *일치(match)*의 종류는 다음과 같습니다.

- 맨 처음에만.

  DB2 외부에서 유지보수되는 문서 응용프로그램으로부터 문자열 `myocardial infarction`이 들어 있는 모든 문서 ID의 목록을 검색합니다. 검색은 비용이 많이 소모되는 프로세스이므로, 함수는 한 번만 수행하고자 하며, 이 목록을 후속 호출에서 사용하기 편한 어딘가에 저장하려 합니다.

- 매 호출시.

  첫번째 호출 중에 저장된 문서 ID 목록을 사용하여 두 번째 인수에 전달된 문서 ID가 목록에 들어 있는지 보십시오.

이 특정 *일치(match)* 함수는 DETERMINISTIC 또는 NOT VARIANT입니다. 이것의 응답은 입력 인수 값에만 의존합니다. 여기에 나와 있는 것은 올바름이 아닌 성능이 하나의 호출에서 다음 호출로 정보를 저장하는 능력에 의존하는 함수입니다.

이들 모두의 필요는 CREATE FUNCTION문에 SCRATCHPAD를 지정하는 기능으로 충족됩니다.

```
CREATE FUNCTION counter()
  RETURNS int ... SCRATCHPAD;
CREATE FUNCTION match(varchar(200), char(15))
  RETURNS char(1) ... SCRATCHPAD;
```

이 SCRATCHPAD 키워드는 DB2에게 함수에 대한 스크래치 패드를 할당하고 유지보수하도록 알려줍니다. DB2는 2진 0으로 스크래치 패드를 초기화합니다. 테이블 함수가 기본값인 NO FINAL CALL과 함께 지정되는 경우, DB2는 각 OPEN 호출 전에 스크래치 패드를 새로 고칩니다. 테이블 함수 옵션 FINAL CALL을 지정하는 경우, DB2는 그 이후 스크래치 패드의 내용을 검사하거나 변경하지 않습니다. 스크래치 패드는 매번 호출될 때마다 함수에 전달됩니다. 함수는 다시 입력될 수 있으며, DB2는 상태 정보를 스크래치 패드에 보관합니다.

그러므로 *계수기(counter)* 예의 경우 리턴된 마지막 값은 스크래치 패드에 보관될 수 있습니다. 그리고 *일치(match)* 예의 경우, 스크래치 패드가 충분히 크면 문서

목록을 스크래치 패드에 보관할 수 있고, 그렇지 않으면 이 목록을 위한 메모리를 할당하고 스크래치 패드에 확보한 메모리 주소를 보관할 수 있습니다.

UDF가 시스템 자원을 확보하고자 할 수도 있으므로 UDF는 FINAL CALL 키워드를 사용하여 정의될 수 있습니다. 이 키워드는 DB2에게 알려 명령문의 끝 처리에서 UDF를 호출하여 UDF가 시스템 자원을 릴리스할 수 있도록 합니다. 특히, 스크래치 패드가 고정된 크기이므로, UDF는 자기 자신을 위한 메모리를 할당하려 할 수 있으며, 이렇게 하여 최종 호출을 사용하여 메모리를 해제합니다. 예를 들어 위의 *일치(match)* 함수는 얼마나 많은 문서들이 주어진 텍스트 문자열과 일치할 것인지를 예측하지 못합니다. 그러므로 *일치(match)*에 대한 더 나은 정의는 다음과 같습니다.

```
CREATE FUNCTION match(varchar(200), char(15))
  RETURNS char(1) ... SCRATCHPAD  FINAL CALL;
```

스크래치 패드를 사용하고 부속 조회에서 참조되는 UDF의 경우, DB2는 최종 호출을 수행하고(UDF에 그렇게 지정되어 있는 경우) 부속 조회 호출과 호출 사이에 스크래치 패드를 갱신하기로 결정할 수 있습니다. 사용자의 UDF가 부속 조회에서 사용되는 경우, FINAL CALL을 사용하여 UDF를 정의하고 호출 유형 인수를 사용하거나 항상 *2진 0* 조건을 점검함으로써 이러한 가능성을 방지할 수 있습니다.

FINAL CALL을 지정한 경우, 사용자의 UDF가 FIRST 유형의 호출을 수신 여부에 주의하십시오. 이것은 일부 지속적인 자원을 확보하고 초기화하는 데 사용될 수 있습니다.

테이블 함수 고려사항

외부 테이블 함수는 테이블을 참조되는 SQL로 전달하는 UDF입니다. 테이블 함수 참조는 SELECT의 FROM절에서만 유효합니다. 테이블 함수를 사용할 때는 다음에 주의하십시오.

- 테이블 함수가 테이블을 전달한다고 하더라도, DB2와 UDF간의 실제 인터페이스는 한 번에 한 행씩입니다. 테이블 함수에 수행되는 호출에는 OPEN, FETCH, CLOSE, FIRST 및 FINAL의 5가지 유형이 있습니다. FIRST와 FINAL 호

출은 사용자가 UDF를 정의하는 방식에 따라 있을 수도 있고 없을 수도 있습니다. 스칼라 함수에 사용될 수 있는 동일한 호출 *유형* 메카니즘이 이들 호출을 구별하는 데 사용됩니다.

- DB2와 사용자 정의 스칼라 함수 간에 사용되는 표준 인터페이스가 확장되어 테이블 함수를 수용합니다. *SQL-result* 인수가 테이블 함수에 대하여 반복되며, 각 인스턴스는 CREATE FUNCTION문의 RETURNS TABLE절에 정의된대로 리턴되는 컬럼에 대응됩니다. *SQL-result-ind* 인수도 마찬가지로 반복되며, 각 인스턴스는 해당 *SQL-result* 인스턴스와 관련됩니다.
- 테이블 함수의 CREATE FUNCTION문의 RETURNS절에 정의된 모든 결과 컬럼이 리턴될 필요는 없습니다. CREATE FUNCTION의 DBINFO 키워드와 해당 *dbinfo* 인수를 사용하여 특정 테이블 함수 참조에 필요한 컬럼만 리턴되도록 하는 최적화가 가능합니다.
- 리턴된 개별적인 컬럼 값은 형식에 있어서 스칼라 함수에 의해 리턴된 값과 일치합니다.
- 테이블 함수의 CREATE FUNCTION문에는 *CARDINALITY n* 스펙이 있습니다. 이 스펙을 사용하여 정의자는 DB2 최적화 알고리즘에게 대략적인 결과 크기를 알려줌으로써, 최적화 알고리즘이 함수가 참조될 때 더 나은 결정을 내릴 수 있도록 합니다.

  테이블 함수의 CARDINALITY로 무엇이 지정되었는지에 관계없이, 무한 기본 행수(cardinality)를 갖는 함수를 작성할 때, 즉 FETCH 호출시 항상 한 행을 리턴하는 함수를 작성할 때는 주의가 필요합니다. DB2가 조회 처리 내 촉매로서 *end-of-table* 조건을 기대하는 상황은 많습니다. GROUP BY나 ORDER BY를 사용하는 것이 이러한 경우의 예입니다. DB2는 *end-of-table*에 도달해야지만 총계를 위한 그룹을 형성할 수 있으며, 데이터를 모두 갖게 되어야 정렬할 수 있습니다. 그러므로 테이블 끝 조건(SQL-state 값 ’02000’)을 리턴하지 않는 테이블 함수는 GROUP BY나 ORDER BY절과 함께 사용하는 경우, 무한 처리 루프를 발생시킬 수 있습니다.

테이블 함수 오류 처리

테이블 함수 호출의 오류 처리 모델은 다음과 같습니다.

1. FIRST 호출이 실패하면, 더 이상의 호출이 수행되지 않습니다.

2. FIRST 호출이 성공하면 중첩된 OPEN, FETCH 및 CLOSE 호출이 이루어지고, FINAL 호출이 항상 수행됩니다.
3. OPEN 호출이 실패하면 FETCH나 CLOSE 호출이 수행되지 않습니다.
4. OPEN 호출에 성공하면, FETCH 및 CLOSE 호출이 수행됩니다.
5. FETCH 호출에 실패하면, 더 이상의 FETCH 호출은 수행되지 않지만, CLOSE 호출이 수행됩니다.

주: 이 모델은 스칼라 UDF에 대한 일반적인 오류 처리를 설명합니다. 시스템 실패 또는 통신상의 문제가 발생한 경우, 오류 처리 모델에 의한 호출은 수행되지 않을 수도 있습니다. 예를 들어 FENCED UDF의 경우, `db2udf` 분리(fenced) 프로세스가 중간에서 종료한 경우, DB2는 지시된 호출을 수행할 수 없습니다.

스칼라 함수 오류 처리

FINAL CALL 스펙을 사용하여 정의되는 스칼라 UDF에 대한 오류 처리 모델은 다음과 같습니다.

- FIRST 호출이 실패하면, 더 이상의 호출이 수행되지 않습니다.
- FIRST 호출이 성공하면, 명령문 처리에 의해 보장되는 NORMAL 호출이 수행되고, FINAL 호출이 수행됩니다.
- FETCH 호출에 실패하면, 더 이상의 NORMAL 호출은 수행되지 않지만, FINAL CALL을 지정한 경우 FINAL 호출이 수행됩니다.

이것은 FIRST 호출에서 오류가 리턴된 경우, FINAL 호출이 수행되지 않으므로 UDF가 리턴하기 전에 정리해야 함을 의미합니다.

테이블 함수의 오류 처리 모델은 490 페이지의『테이블 함수 고려사항』 절에 정의되어 있습니다.

주: 이 모델은 스칼라 UDF에 대한 일반적인 오류 처리를 설명합니다. 시스템 실패 또는 통신상의 문제가 발생한 경우, 오류 처리 모델에 의한 호출은 수행되지 않을 수도 있습니다. 예를 들어 FENCED UDF의 경우, `db2udf` 분리(fenced) 프로세스가 중간에서 종료한 경우, DB2는 지시된 호출을 수행할 수 없습니다.

UDF 매개변수 또는 결과로 LOB 위치 지정자 사용

CREATE FUNCTION문에서 어떤 LOB 데이터 유형이나 LOB 유형에 근거한 구별 유형에도 AS LOCATOR를 추가할 수 있습니다. 이것은 전달된 매개변수와 리턴된 결과 모두에 적용됩니다. 이러한 경우 DB2는 다음을 수행합니다.

- 매개변수의 경우, DB2는 전체 LOB 값 대신 4 바이트의 위치 지정자를 전달합니다. 이 위치 지정자는 실제 바이트를 검색하고 조작하기 위한 특별한 API 세트와 관련되어 여러 가지 방법으로 사용될 수 있습니다. UDF가 값의 단지 몇 바이트만을 필요로 하는 경우, 저장된 것은 지워집니다.

  저장영역 전체 LOB를 위한 메모리가 할당될 필요가 없습니다.

  성능 전체 값을 결과화하는 것은 막대한 입출력 시간과 바이트 이동 명령을 소모하게 됩니다.

- 결과의 경우, 전체 LOB 값 대신 4바이트의 위치 지정자를 리턴합니다. 이 경우에도 저장영역과 성능상의 이점이 발생합니다.

위치 지정자 값을 수정하는 경우, 사용할 수 없게 되므로 수정하지 마십시오. API가 오류를 리턴할 것입니다.

이 특수한 API들은 NOT FENCED로 정의된 UDF에서만 사용할 수 있습니다. 이것은 버그가 있는 UDF가 시스템을 손상시킬 수 있는 가능성 때문에, 테스트 단계의 이들 UDF가 실제 데이터베이스에서 사용되어서는 안됨을 의미합니다. 테스트 데이터베이스에서 수행될 때, 버그가 있는 경우, UDF로부터 지속적인 손상이 발생될 수는 없습니다. UDF에 오류가 없는 것으로 판명되면, 이것은 실제 데이터베이스에 적용될 수 있습니다.

다음에 나오는 API들은 sqludf.h UDF include 파일에 들어 있는 함수 프로토타입을 사용하여 정의된 것입니다.

```
extern int sqludf_length(
    sqludf_locator*     udfloc_p,       /* in:  User-supplied LOB locator value */
    sqlint32*           Return_len_p    /* out: Return the length of the LOB value */
);
extern int sqludf_substr(
    sqludf_locator*     udfloc_p,       /* in:  User-supplied LOB locator value */
    sqlint32            start,          /* in:  Substring start value (starts at 1) */
    sqlint32            length,         /* in:  Get this many bytes */
    unsigned char*      buffer_p,       /* in:  Read into this buffer */
    sqlint32*           Return_len_p    /* out: Return the length of the LOB value */
);
extern int sqludf_append(
    sqludf_locator*     udfloc_p,       /* in:  User-supplied LOB locator value */
```

```
    unsigned char*      buffer_p,       /* in:  User's data buffer */
    sqlint32            length,         /* in:  Length of data to be appended */
    sqlint32*           Return_len_p    /* out: Return the length of the LOB value */
);
extern int sqludf_create_locator(
    int                 loc_type,       /* in:  BLOB, CLOB or DBCLOB? */
    sqludf_locator**    Loc_p           /* out: Return a ptr to a new locator */
);
extern int sqludf_free_locator(
    sqludf_locator*     loc_p           /* in: User-supplied LOB locator value */
);
```

다음은 이 API들이 조작되는 방식에 대한 설명입니다. 모든 길이는 데이터 유형에 관계없이 바이트 단위이고, 1 바이트 또는 2 바이트 문자와도 관계 없습니다.

리턴 코드. 각 API에 대하여 DB2에 의해 UDF로 전달되는 리턴 코드는 다음과 같이 해석됩니다.

0 성공.

-1 API로 전달된 위치 지정자가 호출하기 전에 `sqludf_free_locator()`에 의해 해제되었습니다.

-2 FENCED 모드 UDF에서 호출이 시도되었습니다.

-3 API에 불량 입력 값이 제공되었습니다. 각 API에 고유한 불량 입력 값의 예는 아래 설명을 참조하십시오.

기타 유효하지 않은 위치 지정자 또는 메모리 오류와 같은 기타 오류. 이 경우에 리턴되는 값은 해당 오류 조건에 대응하는 SQLCODE입니다. 예를 들어, -423은 유효하지 않은 위치 지정자 값을 의미합니다. 이러한 "기타" 코드 중 하나를 가지고 UDF로 리턴하기 전에 DB2는 오류의 심각도에 대해 판단합니다. 심각한 오류의 경우, DB2는 오류가 발생했음을 기억하고, UDF가 DB2로 리턴할 때 UDF가 DB2로 오류 SQLSTATE를 리턴하는지 여부에 관계없이, DB2는 오류 조건에 적절한 조치를 취합니다. 심각하지 않은 오류의 경우, DB2는 오류가 발생했다는 것을 잊고, 정정 조치를 취할 것인지 또는 DB2로 오류 SQLSTATE를 리턴할 것인지 여부를 UDF가 결정하도록 맡깁니다.

- `sqludf_length()`.

  LOB 위치 지정자가 주어진 경우, 위치 지정자에 의해 표시되는 LOB 값의 길이를 리턴합니다. 문제가 되는 위치 지정자는 보통 DB2에 의해 UDF로 전달되는 위치 지정자이지만, UDF에 의해 `sqludf_append()`를 사용하여 구축되는 결과 값을 나타내는 위치 지정자일 수 있습니다.

일반적으로 UDF가 위치 지정자를 수신할 때 LOB 값의 길이를 알고자 하는 경우 UDF는 이 API를 사용합니다.

리턴 코드 **3**은 다음을 나타냅니다.

– udfloc_p(위치 지정자의 주소)가 0

– return_len_p(길이를 놓을 장소의 주소)가 0

- `sqludf_substr()`

  LOB 위치 지정자, LOB 내의 시작 위치, 원하는 길이 및 버퍼에 대한 포인터 등이 주어진 경우, 이 API는 버퍼에 바이트를 위치시키고, 이동할 수 있는 바이트 수를 리턴합니다. 물론 UDF는 원하는 길이에 충분한 버퍼를 제공해야 합니다. 이동된 바이트 수는 원하는 길이보다 짧을 수도 있는데, 예를 들어, 위치 101에서 시작하는 50 바이트를 요청한 경우, LOB 값은 단지 120 바이트 길이이고 API는 20 바이트만을 이동시킬 것입니다.

  일반적으로 이것은 UDF가 위치 지정자를 수신할 때, LOB 값의 바이트를 보고자하는 경우 UDF가 사용하는 API입니다.

  리턴 코드 **3**은 다음을 나타냅니다.

  – udfloc_p(위치 지정자의 주소)가 0

  – start가 1보다 작음

  – length가 음수

  – buffer_p(버퍼 주소)가 0

  – return_len_p(길이를 놓을 장소의 주소)가 0

- `sqludf_append()`

  LOB 위치 지정자, 안에 데이터가 들어 있는 데이터 버퍼에 대한 포인터 및 추가할 데이터 길이 등이 주어졌을 때, 이 API는 데이터를 LOB 값의 끝에 추가하고 추가된 바이트 길이를 리턴합니다. 추가된 길이는 항상 추가하기 위해 주어진 길이와 같습니다. 전체 길이를 추가할 수 없는 경우, `sqludf_append()` 호출은 기타 리턴 코드와 함께 실패하게 됩니다.

  일반적으로 이것은 결과가 AS LOCATER를 사용하여 정의될 때 그리고 UDF가 `sqludf_create_locator()`를 사용하여 위치 지정자를 작성한 후 한 번에 하나의 추가를 구축할 때 UDF가 사용하는 API입니다. 이 경우 구축 프로세스를 완료한 후 UDF는 위치 지정자를 결과 인수가 가리키는 곳으로 이동시킵니다.

이 API를 사용하여 사용자의 입력 위치 지정자에도 추가할 수 있는데, 이것은 UDF 내에서 사용자 값을 조작할 수 있는 최대한의 융통성을 제공한다는 점에서 매우 유용하지만, SQL문이나 데이터베이스에 저장된 LOB 값에는 아무런 영향을 미치지 않습니다.

이 API는 조금씩 매우 큰 LOB 값을 구축하는 데 사용될 수 있습니다. 결과를 구축하는 데 많은 수의 추가가 사용되는 경우, 이 타스크의 성능은 다음을 수행함으로써 향상될 수 있습니다.

- 큰 응용프로그램 제어 힙(heap) 할당. APP_CTL_HEAP_SZ는 데이터베이스 관리 프로그램 구성 매개변수입니다.
- 더 큰 버퍼의 더 적은 수의 추가를 수행합니다. 예를 들어 50 바이트짜리 20개의 추가보다는 하나의 1000 바이트짜리 추가를 수행합니다.

sqludf_append() API를 사용하여 많은 수의 큰 LOB 값을 구축하는 SQL 응용프로그램들은 사용할 수 있는 디스크 공간의 양에 있어서 제한을 받으므로 오류가 발생할 수 있습니다. 이러한 오류가 발생할 수 있는 가능성은 다음을 수행함으로써 감소할 수 있습니다.

- 개별 추가에 더 큰 버퍼 사용.
- 명령문 간의 빈번한 COMMIT 사용.
- SELECT문의 각 행이 이 API를 통해 LOB 값을 구축하는 경우, 행 간에 CURSOR WITH HOLD를 사용하고 COMMIT을 수행.

리턴 코드 **3**은 다음을 나타냅니다.

- udfloc_p(위치 지정자의 주소)가 0
- length가 음수
- buffer_p(버퍼 주소)가 0

• `sqludf_create_locator()`

예를 들어 `SQL_TYP_CLOB`와 같은 데이터 유형이 주어진 경우, 위치 지정자를 작성합니다. 데이터 유형 값은 외부 응용프로그램 헤더 파일인 `sql.h`에 정의되어 있습니다.

일반적으로 UDF는 UDF 결과가 AS LOCATOR를 사용하여 정의되고, UDF가 `sqludf_append()`를 사용하여 결과 값을 구축하고자 할 때 이 API를 사용합니다. 또 다른 사용법은 내부적으로 LOB 값을 조작할 때 사용합니다.

리턴 코드 **3**은 다음을 나타냅니다.

– udfloc_p(위치 지정자의 주소)가 0

– loc_type은 세 가지 유효한 값 중 하나가 아닙니다.

– loc_p(위치 지정자가 놓일 위치의 주소)가 0입니다.

- `sqludf_free_locator()`

  전달된 위치 지정자를 해제합니다.

  sqludf_create_locator() API를 사용하여 작성되었고, 내부 조작에만 사용된 위치 지정자를 해제할 때 이 API를 사용하십시오. UDF로 전달된 위치 지정자는 해제할 필요가 없습니다. 해당 위치 지정자가 출력으로 UDF로부터 전달되는 경우, sqludf_create_locator()를 통해 UDF에 의해 작성된 위치 지정자는 해제할 필요가 없습니다.

  리턴 코드 **3**은 다음을 나타냅니다.

  – udfloc_p(위치 지정자의 주소)가 0

다음의 주의사항은 이들 API 사용시 적용됩니다.

주:

1. LOB 위치 지정자를 리턴하도록 정의된 UDF에는 몇 가지 가능성을 가지고 있습니다. 이것은 다음을 리턴합니다.
   - 이것으로 전달된 입력 위치 지정자
   - sqludf_append()를 통해 추가된 입력 위치 지정자
   - sqludf_create_locator()를 통해 작성되고 sqludf_append()를 통해 추가된 위치 지정자.
2. 테이블 함수는 하나 이상의 LOB 위치 지정자를 리턴하도록 정의할 수 있습니다. 이들 각각은 앞의 항목에서 논의된 가능성을 가지고 있습니다. 또한 그러한 테이블 함수에 대하여 여러개의 테이블 함수 컬럼에 대한 출력으로 같은 위치 지정자를 리턴하는 것도 유효합니다.
3. 입력 인수로 테이블 함수로 전달된 LOB 위치 지정자는 행 생성 프로세스 내내 살아 있습니다. 실제로, 테이블 함수는 한 행을 생성하는 동안 그러한 LOB 위치 지정자를 사용하여 LOB에 추가하고 추가된 바이트를 그 다음 행에서 볼 수 있습니다.

4. UDF(테이블 또는 스칼라 함수)로부터의 LOB 위치 지정자 출력으로 DB2에서 시작된 LOB를 나타내기 위한 내부 제어 메카니즘은 1950 바이트를 차지합니다. 이러한 이유와 정렬에 대한 입력이 되는 행 크기의 한계로 인해, UDF LOB 위치 지정자로서 시작되는 여러개의 그러한 LOB를 정렬하려는 조회는 연관된 다른 컬럼의 크기에 따라 한 행에 (최대) 두 개의 값으로 제한됩니다. 같은 제한이 테이블에 삽입되는 행에 적용됩니다.

LOB 위치 지정자 사용에 대한 시나리오

이것은 LOB 위치 지정자의 유용함을 보여주는 몇 가지 가능한 시나리오에 대한 간략한 요약입니다. 이 네 개의 시나리오는 위치 지정자의 사용을 대략적으로 보여주며, 요구되는 공간을 절약하고 효율성을 증가시킬 수 있는 방법을 보여줍니다.

- 입력 LOB의 부분들에 대한 다양한 액세스

  UDF는 sqludf_substr()을 사용하여 LOB 값의 처음 부분을 보고, 여기에서 발견한 크기 변수에 근거하여 1억 바이트 LOB 값의 어딘가에서 다시 sqludf_substr()을 사용하여 몇 바이트만을 읽고자 할 수 있습니다.

- 한 번에 입력 LOB 한 부분을 대부분 처리

  이 UDF는 LOB 값에서 무언가를 찾습니다. 대부분은 앞 쪽에서 찾게 되지만, 때로는 1억 바이트 값 전체를 스캔해야 할 수도 있습니다. 이 UDF는 sqludf_length()를 사용하여 이 특정 값의 크기를 찾아, 루프에서 sqludf_substr()에 대한 호출을 지정하여 한 번에 1 000 바이트씩 단계적으로 탐색합니다. 여기에서 시작 위치로 변수를 사용하는데, 이 변수는 루프를 반복할 때마다 매번 1 000씩 증가합니다. 찾고자 하는 것을 찾을 때까지 이런 식으로 계속됩니다.

- 두 개의 입력 LOB 중 하나를 리턴

  이 UDF는 입력으로 두 개의 LOB 위치 지정자를 갖으며, 출력으로 LOB 위치 지정자를 리턴합니다. 이것은 sqludf_substr()을 사용하여 수신한 바이트를 읽어서 두 개의 입력을 검토하고 비교한 후, 특정 알고리즘에 근거하여 둘 중 어떤 것을 선택할 것인지를 판별합니다. 이것을 결정할 때, 선택된 입력의 위치 지정자를 UDF 결과 인수에 의해 나타나는 버퍼로 복사합니다.

- 입력 LOB를 잘라내기 및 붙여넣기를 하고 그 결과를 리턴

이 UDF는 LOB 값과 진행 방식을 알려주는 일부 다른 인수들을 전달받습니다. 이것은 출력을 위해 위치 지정자를 작성하고, 다른 입력 인수에 들어 있는 지시사항에 근거하여 `sqludf_substr()`을 사용하여 읽어 들이는 입력 LOB의 서로 다른 여러 부분으로부터 대부분의 결과 값을 받아 순차적으로 출력 값을 구축합니다. 모두 완료되면, 결과 위치 지정자를 UDF 결과 인수가 가리키는 버퍼로 복사하고 종료합니다.

기타 코딩시 고려사항

이 절에서는 UDF를 구현할 때 추가적으로 고려해야 할 사항, 명심해야 할 항목 및 피해야 할 항목 등을 설명합니다.

도움말 및 추가 정보

다음은 사용자의 UDF를 성공적으로 구축하기 위해 고려해야 하는 권장사항입니다.

- **UDF 내용은 보호되어야 합니다.** 실행 가능한 함수 내용은 DB2에 의해 어떤 방법으로든 캡쳐되지 않거나 보호되어야 합니다. CREATE FUNCTION문은 단지 내용을 가리킵니다. 해당 함수에 의존하는 함수와 데이터베이스 응용프로그램의 무결성을 보존하려면, 사용자는 해당 함수가 들어 있는 디렉토리로의 액세스를 관리하고, 내용 자체를 보호함으로써 함수 내용이 실수로 또는 고의로 삭제되거나 바뀌지 않도록 해야 합니다.
- DB2는 DB2와 SQL(즉, 모든 SQL 인수 및 함수 리턴 값)간의 인터페이스에 모든 버퍼에 대한 포인터를 전달합니다. UDF 인수는 반드시 포인터로 정의해야 합니다.
- 모든 SQL 인수 값이 버퍼에 들어갑니다. 이것은 해당 값의 사본을 만들어 이것이 UDF에 제시됨을 뜻합니다. UDF가 입력 매개변수를 변경하면, 변경된 내용은 SQL 값이나 처리에 아무런 영향을 미치지 않지만 DB2가 제대로 작동하지 않을 수도 있습니다.
- OLE 자동의 경우, 입력 매개변수를 변경하지 마십시오. 그렇지 않으면 메모리 자원이 해제되지 않아 메모리 부족이 발생할 수 있습니다.

주요 OLE 라이브러리 버전이 불일치하거나 OLE 라이브러리 초기화에 실패한 경우, 데이터베이스 관리 프로그램은 SQLCODE -465(SQLSTATE 58032)와 이유 코드 34(OLE 라이브러리 초기화 실패)를 리턴합니다.

- 모든 운영 플랫폼에서의 UDF에서 재진입(re-entrancy)은 **강력히 권장되는** 것으로, 이것의 사본 하나가 복수의 동시 수행 명령문과 응용프로그램에 사용될 수 있습니다.

  SCRATCHPAD 기능은 재진입에 의해 부과되는 많은 제한점을 방지하는 데 사용됩니다.

- 예를 들어, 재컴파일되고 재링크되어 현재 사용되고 있는 함수의 내용이 수정되는 경우, DB2는 트랜잭션 중에 *함수를 변경하지* 않습니다. 그러나 후속 트랜잭션에서 사용되는 사본은, 이러한 종류의 동적 수정이 발생하는 경우 달라질 수 있습니다. 사용자의 운영 체제 역시 사용자가 사용 중인 UDF 내용을 변경하지 못하도록 할 수 있습니다. 이러한 방법은 권장되지 않습니다.

- UDF에서 동적 메모리를 할당하는 경우, DB2로 리턴하기 전에 이것은 해제되어야 합니다. 이것은 특히 NOT FENCED 경우에 중요합니다. 그러나 호출과 호출간에 걸친 UDF에 의해 필요한 동적 메모리를 앵커(anchor)하기 위해서는 SCRATCHPAD 기능이 사용될 수 있습니다. 이러한 방식으로 스크래치 패드를 사용하는 경우, UDF의 CREATE FUNCTION에 FINAL CALL 속성을 지정하십시오. 이렇게 하는 이유는 UDF를 반복해서 사용함으로써, 시간이 지남에 따라 시스템의 메모리가 부족하게 될 수 있기 때문입니다.

  이러한 이유는 UDF가 사용하는 다른 시스템 자원에 대해서도 적용됩니다.

- 필요에 따라 NOT NULL CALL 옵션을 사용하십시오. CREATE FUNCTION 옵션을 사용하면 각각의 SQL 인수가 널(NULL)인지 여부를 점검할 필요가 없고, NULL 값을 가지고 있을 때 더 좋은 성능을 나타냅니다.

- 사용자의 UDF로부터 생성된 결과가 입력 SQL 인수 이외의 다른 것에 의존하는 경우, NOT DETERMINISTIC 옵션을 사용하십시오. 이 옵션은 SQL 컴파일러가 일관성이 없는 결과를 발생시킬 수도 있는 특정 최적화를 수행하지 않도록 합니다.

- 사용자의 UDF에 안정적으로 수행될 필요가 있는 부가작용이 있는 경우, EXTERNAL ACTION 옵션을 사용하십시오. EXTERNAL ACTION은 SQL 컴파일러가 특정 상황에서 사용자의 UDF 호출을 막을 수도 있는 특정 최적화를 수행하지 못하도록 합니다.
- FENCED 및 UNFENCED에서 하나를 선택하는 것에 대하여:

FENCED UDF

FENCED UDF는 자체 프로세스에서 실행하므로 대부분의 DB2 내부 제어와 데이터 영역을 실수나 고의로 액세스할 수 없습니다. 그러므로 데이터베이스에 대해 FENCED UDF를 선택하는 것이 더 안전합니다. 그러나, NOT FENCED UDF만큼 쉽지는 않지만 프로그래밍 오류가 들어 있는 FENCED UDF가 DB2를 종료시킬 가능성은 아직도 있습니다. 예를 들어, 리턴 변수의 다량 겹쳐쓰기를 수행하는 UDF는 DB2를 이상 종료시킬 수 있습니다.

UNFENCED UDF

NOT FENCED UDF는 DB2 엔진 프로세스에 직접 로드되고 실행되므로 FENCED UDF보다 성능이 낫습니다. NOT FENCED UDF는 프로세스 통신 오버헤드의 성능 비용을 회피합니다. 그러나 NOT FENCED UDF는 DB2 내부 제어나 데이터 영역을 액세스하거나 변경할 수 있습니다. FENCED UDF보다는 제대로 작성되지 않은 NOT FENCED UDF로 DB2를 더 쉽게 종료시킬 수 있습니다.

FENCED 및 NOT FENCED UDF 둘다에 대해 확실하게 다음과 같이 수행해야 합니다.

- UDF가 견고하게 작성되었는지 확인합니다
- UDF를 엄격히 설계하고 그 코드를 검토합니다
- UDF가 제대로 작성되지 않은 경우 테스트 데이터베이스와 같이 아무런 피해를 입히지 않는 환경에서 UDF를 테스트합니다.

UDF에 의해 일어나는 대부분의 이상 종료는 -430 SQLCODE를 리턴하고 데이터베이스 손상을 방지하는 DB2에서 감지됩니다. 그러나, 대량 리턴 값 버퍼의 겹쳐쓰기를 포함한 특정 UDF 오동작은 UDF뿐만 아니라 DB2를 실패하게

만들 수 있습니다. 가변 길이 데이터를 리턴하거나 리턴 값 버퍼로 옮겨야 하는 바이트 수를 계산하는 UDF에 특히 주의하십시오.

- EUC 코드 세트를 사용하는 UDF 사용시 고려사항은 585 페이지의 『UDF에 대한 고려사항』의 내용을 참조하십시오.
- NOT FENCED UDF를 수행하는 응용프로그램의 경우, 맨 처음 이러한 UDF가 호출될 때 UDF_MEM_SZ 구성 매개변수에 의해 표시된 *크기*의 메모리 블록이 작성됩니다. 그리고나서 명령문별로 DB2와 NOT FENCED UDF간의 인터페이스를 위한 메모리가 필요에 따라 이 메모리 블록으로부터 할당되고 할당해제됩니다.

  FENCED UDF의 경우, 다른 메모리 블록이 같은 방법으로 사용됩니다. 메모리가 프로세스들 간에 공유되기 때문에 이것은 다릅니다. 실제로, 응용프로그램이 NOT FENCED와 FENCED UDF를 모두 사용하는 경우, UDF_MEM_SZ 매개변수로 각각의 *크기*가 표시되는 두 개의 별도의 메모리 블록이 사용됩니다. 이 구성 매개변수에 대해서는 *관리 안내서*에서 자세한 내용을 참조하십시오.
- 다음 상황에서는 DISALLOW PARALLELISM 옵션을 사용하십시오.
  - 스칼라 UDF에서, 사용자의 UDF가 절대적으로 같은 사본을 수행하는 경우. 일반적으로 이것은 NOT DETERMINISTIC SCRATCHPAD UDF의 경우입니다. (예를 들어, 488 페이지의 『스크래치 패드 고려사항』에 지정된 계수기(counter) UDF를 보십시오.)
  - UDF가 단일 참조를 위해 한 번에 복수의 파티션에서 수행되는 것을 원하지 않는 경우.
  - 테이블 함수를 지정하는 경우.

  그렇지 않은 경우, 기본값인 ALLOW PARALLELISM을 지정해야 합니다.

UDF 제한사항 및 조건

이 절에서는 사용자의 UDF에서 피해야 할 항목에 대하여 설명합니다.

1. 일반적으로 DB2는 운영 체제 기능 사용을 제한하지 않습니다. 그러나 몇 가지 예외가 있는데 다음과 같습니다.
   a. 신호 또는 예외 핸들러를 등록하는 경우, 이것이 DB2가 같은 핸들러를 사용하는 것을 방해하여 예상치 못한 장애가 발생할 수 있습니다.

b. 프로세스를 종료하는 시스템 호출은 DB2의 프로세스 중 하나를 이상 종료시키고 시스템이나 응용프로그램 장애를 발생시킬 수 있습니다.

   다른 시스템 호출의 경우, 이들이 DB2의 정상적인 조작을 방해하는 경우 문제가 발생할 수 있습니다. 예를 들어 UDF가 들어 있는 라이브러리를 메모리에서 로드 해제하려는 UDF는 심각한 문제를 발생시킬 수 있습니다. 시스템 호출이 들어 있는 UDF를 코딩 및 테스트하는 경우 주의해 주십시오.

2. 'DB2'로 시작하는 모든 환경 변수의 값은 데이터베이스 관리 프로그램이 `db2start`를 사용하여 시작될 때 캡쳐되며, FENCED 되었는지 여부에 관계없이 모든 UDF에서 사용할수 있습니다. 유일한 예외는 `DB2CKPTR` 환경 변수입니다. 환경 변수가 *캡쳐된다*는점에 주의하십시오. `db2start`가 발행된 이후에 환경 변수의 변경된 내용은 UDF에서 사용할 수 없습니다.
3. 외부 UDF로 전달된 LOB의 경우, *UDF 공유 메모리 크기*라는 DB2 시스템 구성 매개변수에 의해 지정된 최대 *크기*에 제한을 받습니다. 이 매개변수에 지정할 수 있는 최대 값은 256M입니다. DB2에서의 기본 설정값은 1M입니다. 이 매개변수에 대해서는 *관리 안내서*에서 자세한 내용을 참조하십시오.
4. 화면과 키보드로의 입력 및 출력은 권장되지 않습니다. DB2의 프로세스 모델에서 UDF는 백그라운드에서 수행되어 화면에서 기록할 수 없습니다. 그러나 파일에 기록할 수 있습니다.

   주: DB2는 DB2 자신의 트랜잭션을 사용하여 UDF에 의해 수행되는 외부 입/출력을 동기화하려고 시도하지 않습니다. 예를 들어, UDF가 트랜잭션 동안에 파일에 기록하고, 이 트랜잭션이 어떤 이유에서인지 나중에 백 아웃된 경우, 이 파일에 대한 기록을 발견하거나 실행취소하려는 시도가 이루어지지 않습니다.

5. UNIX 기반 시스템에서, 사용자의 UDF는 DB2 에이전트 프로세스(NOT FENCED)의 사용자 ID나 `db2udf` 실행가능 파일(FENCED)을 소유하는 사용자 ID의 사용자 ID에서 실행합니다. 이 사용자 ID는 UDF에서 사용할 수 있는 시스템 자원을 제어합니다. `db2udf` 실행가능 파일에 대한 정보는 사용자 플랫폼의 *빠른 시작* 책을 참조하십시오.

6. UDF 내에서 보호된 자원 즉, 한 번에 하나의 프로세스 액세스만을 허용하는 자원을 사용할 때, UDF 간의 교착 상태를 피하도록 해야 합니다. 둘 또는 그 이상의 UDF가 교착상태에 빠지면, DB2는 이 조건을 탐지하지 못하게 됩니다.

7. 문자 데이터는 데이터베이스의 코드 페이지로 외부 함수로 전달됩니다. 마찬가지로 이 함수의 출력인 문자열은 데이터베이스의 코드 페이지를 사용하는 것으로 데이터베이스는 간주합니다. 응용프로그램 코드 페이지가 데이터베이스 코드 페이지와 다른 경우, SQL문에서 다른 값에 대하여 그러하듯이 코드 페이지 변환이 발생합니다. 사용자의 CREATE FUNCTION문에 문자 매개변수나 결과 속성으로 FOR BIT DATA를 코딩함으로써, 이 변환을 막을 수 있습니다. 문자 매개변수가 FOR BIT DATA 속성을 사용하여 정의되지 않는 경우, 사용자의 UDF 코드는 데이터베이스 코드 페이지의 인수를 수신합니다.

   CREATE FUNCTION에서 DBINFO 옵션을 사용하면, 데이터베이스 코드 페이지가 UDF로 전달됩니다. 이 정보를 사용하여 코드 페이지에 민감한 UDF는 많은 여러 코드 페이지에서 운영되도록 작성될 수 있습니다.

8. C++를 사용하여 UDF를 작성할 때, 다음과 같이 함수 이름 선언을 고려할 수 있습니다.

   ```
   extern "C" void SQL_API_FN udf( ...arguments... )
   ```

   extern "C"는 C++ 컴파일러에 의해 명명된 함수 이름의 유형 장식(또는 '절단(mangling)')을 허용하지 않습니다. 이러한 선언이 없으면, CREATE FUNCTION문을 발행할 때 함수 이름에 대한 모든 유형 장식(type decoration)을 포함시켜야 합니다.

UDF 코드 예

다음과 같은 예제 UDF 코드가 DB2에서 제공됩니다.

예: 정수 나누기 연산자

예: CLOB 접기, 모음 찾기

예: 계수기(Counter)

제공되는 모든 예들이 있는 위치와 이들을 호출하는 방법은 835 페이지의 『부록 B. 샘플 프로그램』의 내용을 참조하십시오.

UDF 컴파일 및 링크에 관한 정보는 *응용프로그램 빌드 안내서* 책을 참조하십시오.

각각의 UDF 예는 그에 해당되는 CREATE FUNCTION문 및 그 사용을 보여주는 간단한 시나리오와 함께 제공됩니다. 이 시나리오들은 모두 다음의 테이블 TEST를 사용하는데, 이것은 시나리오에서 설명하고자 하는 바를 보여주기 위해 작성된 것입니다. 다음은 테이블 정의입니다.

```
CREATE TABLE TEST (INT1 INTEGER,
                   INT2 INTEGER,
                   PART CHAR(5),
                   DESCR CLOB(33K))
```

테이블에 데이터를 입력한 후, CLP를 사용하여 다음의 명령문을 발행하여 그 내용을 표시하십시오.

```
SELECT INT1, INT2, PART, SUBSTR(DESCR,1,50) FROM TEST
```

출력의 잘 볼 수 있게 하기 위해 CLOB 컬럼에서 사용한 SUBSTR 함수에 주의하십시오. 사용자는 다음과 같은 CLP 출력을 받게 됩니다.

```
INT1        INT2        PART  4
----------- ----------- ----- --------------------------------------------------
         16           1 brain The only part of the body capable of forgetting.
          8           2 heart The seat of the emotions?
          4           4 elbow That bendy place in mid-arm.
             2           0          -          -
         97          16 xxxxx Unknown.
       5개의 레코드가 선택되었습니다.
```

아래의 예와 시나리오를 읽으면서 테이블 TEST의 앞서 나온 정보를 참조하십시오.

예: 정수 나누기 연산자

DB2에서의 정수 나누기 작업에서, 오류 SQLCODE -802(SQLSTATE 22003)이 발생하고, 0으로 나누면 명령문을 종료하기 때문에 사용자는 이러한 정수 나누기 작업 방식에 불만이 있다고 가정합니다. 사용자가 DFT_SQLMATHWARN 구성 매개변수를 사용하는 *친근한* 산술 연산을 작동 가능하게 했다면, DB2는 오류 대신 널(NULL)을 리턴합니다. 대신, 정수 나누기할 때 널(NULL)을 리턴하도록 하기 위해 이 UDF를 코딩합니다.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqludf.h>
#include <sqlca.h>
#include <sqlda.h>

 /*********************************************************************
*  function divid: performs integer divid, but unlike the / operator
*                  shipped with the product, gives NULL when the
*                  denominator is zero.
*
*                This function does not use the constructs defined in the
*                "sqludf.h" header file.
*
*     inputs:  INTEGER num     numerator
*              INTEGER denom   denominator
*     output:  INTEGER out     answer
**********************************************************************/
#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN divid (
   sqlint32 *num,                          /* numerator */
   sqlint32 *denom,                        /* denominator */
   sqlint32 *out,                          /* output result */
   short *in1null,                         /* input 1 NULL indicator */
   short *in2null,                         /* input 2 NULL indicator */
   short *outnull,                         /* output NULL indicator */
   char *sqlstate,                         /* SQL STATE */
   char *funcname,                         /* function name */
   char *specname,                         /* specific function name */
   char *mesgtext) {                       /* message text insert */

   if (*denom == 0) {      /* if denominator is zero, return null result */
      *outnull = -1;
   } else {                /* else, compute the answer */
      *out = *num / *denom;
      *outnull = 0;
   } /* endif */
}
/* end of UDF : divid */
```

이 UDF의 경우 다음에 주의하십시오.

- 여기에는 sqludf.h가 포함되지 않습니다.
- 두 개의 입력 인수가 정의되고 한 개의 출력 인수가 정의됩니다.
- void를 리턴하도록 정의됩니다. 입력 인수를 사용하여 일반적인 UDF 출력이 리턴됨을 명심하십시오.
- 함수 정의에 SQL_API_FN를 포함시키는 것은 여러 플랫폼에 걸쳐 함수 소스의 이식성을 확보하기 위해 설계된 것입니다. 사용자의 UDF 소스 파일에 다음과 같은 명령문을 포함시켜야 합니다.

```
#include <sqlsystm.h>
```

- 아래에서와 같이 CREATE FUNCTION문에 기본값으로 NOT NULL CALL 매개변수가 지정되므로, 널(NULL) 입력 인수에 대한 점검을 하지 않습니다.

다음은 이 UDF에 대한 CREATE FUNCTION문입니다.

```
CREATE FUNCTION MATH."/"(INT,INT)
  RETURNS INT
  NOT FENCED
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME '/u/slick/udfx/div' ;
```

이것은 UDF의 AIX 버전에 대한 것입니다. 다른 플랫폼의 경우 EXTERNAL NAME절에 지정된 값을 수정해야 할 것입니다.

이 명령문의 경우, 다음에 주의하십시오.

- 이것은 MATH 스키마에 정의되어 있습니다. 사용자의 사용자 ID와 같지 않은 스키마에 UDF를 정의하려면, 데이터베이스에 대한 DBADM 권한이 필요합니다.
- 함수 이름은 SQL 나누기 연산자와 같은 이름인 "/"로 정의됩니다. 사실 이 UDF는 A / B와 같이 *삽입 표기*를 사용하여 내장 / 연산자처럼 호출되거나, "/"(A,B)와 같이 *함수 표기*로 호출될 수 있습니다. 다음을 참조하십시오.
- 사용자는 프로그램에 오류가 없음을 확실하게 확신하므로 NOT FENCED로 정의했습니다.

- 사용자는 기본값이 NOT NULL CALL을 사용했는데, 이것을 통해 DB2는 인수 중 어느 하나가 NULL인 경우 함수 내용을 호출하지 않고 NULL 결과를 제공할 수 있습니다.

이제 다음의 두 가지 명령문을 수행하는 경우(CLP 입력이 표시되었음):

```
SET CURRENT FUNCTION PATH = SYSIBM, SYSFUN, SLICK
SELECT INT1, INT2, INT1/INT2, "/"(INT1,INT2) FROM TEST
```

이 출력을 CLP에서 얻게 됩니다(DFT_SQLMATHWARN 데이터베이스 구성 매개변수와의 *친근한 산술*을 허용하지 않는 경우).

```
INT1        INT2        3           4
----------- ----------- ----------- -----------
         16           1          16          16
          8           2           4           4
          4           4           1           1
SQL0802N  산술 오버플로우 또는 다른 산술 예외가 발생했습니다.
SQLSTATE=22003
```

사용자가 CURRENT FUNCTION PATH 특수 레지스터를 "/" UDF가 정의된 스키마인 MATH가 포함되지 않은 스키마들의 병합으로 설정했기 때문에 SQL0802N 오류 메시지가 발생합니다. 그러므로 사용자는 DB2의 내장 나누기 연산자를 실행시키고, 이 연산자의 정의된 기능은 "0으로 나누기" 조건이 발생하면 오류를 발생시킵니다. TEST 테이블의 4번째 행에서 이 조건을 제공합니다.

그러나 함수 경로를 변경하여 경로에서 MATH를 SYSIBM 앞에 놓는 경우, SELECT문을 다시 수행하십시오.

```
SET CURRENT FUNCTION PATH = MATH, SYSIBM, SYSFUN, SLICK
SELECT INT1, INT2, INT1/INT2, "/"(INT1,INT2) FROM TEST
```

그러면 다음의 CLP 출력에서 볼 수 있듯이 원하는 작동 결과를 얻을 수 있습니다.

```
INT1        INT2        3           4
----------- ----------- ----------- -----------
         16           1          16          16
          8           2           4           4
          4           4           1           1
          2           0           -           -
         97          16           6           6
    5개의 레코드가 선택되었습니다.
```

위의 예에서 다음에 주목하십시오.

- SET CURRENT FUNCTION PATH문은 이것이 동적 SQL이고 MATH 스키마를 SYSIBM 앞에 놓기 때문에 다음 명령문에서 사용되는 현재의 함수 경로를 변경합니다.
- 4번째 행은 나누기 결과 NULL를 생성하고, 명령문은 계속 수행됩니다.
- *삽입 구문*과 *접두부 구문* 모두 이 특정 UDF를 호출하는 데 사용될 수 있는데, 이는 그 이름이 내장 연산자와 동일하며, 둘다 위의 예에서 *사용되어* 동일한 결과를 생성하기 때문입니다.
- 실제적으로는 내장 함수와 연산자가 DB2에 정의되는 방식 때문에, "/"가 SMALLINT에 대한 조작에 사용되지 않습니다. DB2 함수 선택 알고리즘은 일치하긴 하지만 정확한 일치는 아닌 이 사용자 정의 "/"보다는 정확한 일치 내장 "/" 연산자를 선택합니다. 외관상 이러한 불일치에 대한 다른 방법이 있습니다. 사용자는 "/"를 호출하기 전에 명시적으로 SMALLINT 인수를 INTEGER로 유형변환할 수 있습니다. 예를 들어, INT1 / INTEGER(SMINT1)에서 컬럼 SMINT1은 SMALLINT로 간주됩니다. 그렇지 않으면, 이보다 더 좋은 방법으로 "/" 연산자를 더 오버로드하는 추가적인 UDF를 등록하는데, 이것은 SMALLINT인 첫번째와 두 번째 매개변수를 정의합니다. 이러한 추가적인 UDF는 MATH."/"입니다.

  이러한 경우, 완전히 일반적인 함수 세트에 대해 정수 나누기를 완벽하게 처리하기 위해 다음과 같은 세 개의 추가적인 함수를 CREATE해야 합니다.

```
CREATE FUNCTION MATH."/"(SMALLINT,SMALLINT)
  RETURNS INT
  SOURCE MATH."/"(INT,INT)

CREATE FUNCTION MATH."/"(SMALLINT,INT)
  RETURNS INT
```

```
    SOURCE MATH."/"(INT,INT)

  CREATE FUNCTION MATH."/"(INT,SMALLINT)
    RETURNS INT
    SOURCE MATH."/"(INT,INT)
```

세 개의 UDF가 추가되어도 추가적인 코드는 MATH."/"입니다.

이제 이러한 네 개의 "/" 함수 정의를 사용하여 정수 나누기의 새로운 기능을 활용하려면, 언제라도 함수 경로에서 SYSIBM 앞에 MATH를 놓고 SQL을 작성하기만 하면 됩니다.

앞의 예에서는 BIGINT 데이터 유형을 고려하지 않았지만, 이 예를 확장시켜 BIGINT를 쉽게 포함시킬 수 있습니다.

예: CLOB 접기, 모음 찾기

사용자의 텍스트 처리 응용프로그램을 보조하기 위한 두 개의 UDF를 코딩했다고 가정하십시오. 첫번째 UDF는 사용자의 *n*번째 이후의 텍스트 문자열을 *접습니다*. 이 예에서 접는다는 것의 의미는 원래 *n* 바이트의 뒤에 있던 부분을 *n+1* 바이트의 앞 부분 앞에 놓는 것입니다. 즉, 이 UDF는 문자열의 맨 처음 n 바이트를 문자열의 맨 뒤로 이동합니다. 두 번째 함수는 텍스트 문자열에서 처음으로 나타나는 모음의 위치를 리턴합니다. 이들 함수 모두 udf.c 예제 파일에 코딩되어 있습니다.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqludf.h>
#include <sqlca.h>
#include <sqlda.h>
#include "util.h"

 /*******************************************************************
*  function fold: input string is folded at the point indicated by the
*                 second argument.
*
*     input: CLOB    in1         input string
*            INTEGER in2         position to fold on
*            CLOB    out         folded string
*******************************************************************/
#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN fold (
   SQLUDF_CLOB     *in1,                /* input CLOB to fold */
   SQLUDF_INTEGER  *in2,                /* position to fold on */
```

```
   SQLUDF_CLOB     *out,               /* output CLOB, folded */
   SQLUDF_NULLIND  *in1null,           /* input 1 NULL indicator */
   SQLUDF_NULLIND  *in2null,           /* input 2 NULL indicator */
   SQLUDF_NULLIND  *outnull,           /* output NULL indicator */
   SQLUDF_TRAIL_ARGS) {                /* trailing arguments */

   SQLUDF_INTEGER len1;

   if (SQLUDF_NULL(in1null) || SQLUDF_NULL(in2null)) {
      /* one of the arguments is NULL.  The result is then "INVALID INPUT" */
      strcpy( ( char * ) out->data, "INVALID INPUT" ) ;
      out->length = strlen("INVALID INPUT");
   } else {
      len1 = in1->length;              /* length of the CLOB */

      /* build the output by folding at position "in2" */
      strncpy( ( char * ) out->data, &in1->data[*in2], len1 - *in2
) ;
      strncpy( ( char * ) &out->data[len1 - *in2], in1->data, *in2
) ;
      out->length = in1->length;
   } /* endif */
   *outnull = 0;                       /* result is always non-NULL */
}
/* end of UDF : fold */

 /*********************************************************************
*  function findvwl: returns the position of the first vowel.
*                 returns an error if no vowel is found
*                 when the function is created, must be defined as
*                 NOT NULL CALL.
*     inputs: VARCHAR(500) in
*     output: INTEGER      out
*********************************************************************/
#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN findvwl (
   SQLUDF_VARCHAR   *in,               /* input character string */
   SQLUDF_SMALLINT  *out,              /* output location of vowel */
   SQLUDF_NULLIND   *innull,           /* input NULL indicator */
   SQLUDF_NULLIND   *outnull,          /* output NULL indicator */
   SQLUDF_TRAIL_ARGS) {                /* trailing arguments */

   short i;                            /* local indexing variable */

   for (i=0; (i < (short)strlen(in) &&        /* find the first vowel */
      in[i] != 'a' && in[i] != 'e' && in[i] != 'i' &&
      in[i] != 'o' && in[i] != 'u' && in[i] != 'y' &&
      in[i] != 'A' && in[i] != 'E' && in[i] != 'I' &&
      in[i] != 'O' && in[i] != 'U' && in[i] != 'Y'); i++);
   if (i == strlen( ( char * ) in )) {               /* no vowels found
*/
            /* error state */
      strcpy( ( char * ) sqludf_sqlstate, "38999" ) ;
            /* message insert */
      strcpy( ( char * ) sqludf_msgtext, "findvwl: No Vowel" ) ;
   } else {                            /* a vowel was found at "i" */
      *out = i + 1;
```

```
      *outnull = 0;
   } /* endif */
}
/* end of UDF : findvwl */
```

위의 UDF의 경우 다음에 주의하십시오.

- 여기에는 sqludf.h가 포함되며, 이 파일에 들어 있는 인수 정의와 마크로를 사용합니다.
- fold() 함수는 NULL 인수를 가지고도 호출되며, 이 경우 INVALID INPUT 문자열을 리턴합니다. 한편 findvwl() 함수는 널 인수를 지정하면 호출할 수 없습니다. sqludf.h에 정의된 SQLUDF_NULL() 마크로를 사용하면 fold()의 널 인수를 점검합니다.
- findvwl() 함수는 오류 SQLSTATE와 메시지 토큰을 설정합니다.
- fold() 함수는 CLOB 데이터 유형을 텍스트 입력 인수로 받아들이는 한편 CLOB 값을 리턴합니다. findvwl()은 VARCHAR 입력 인수를 갖습니다.

다음은 이들 UDF에 대한 CREATE FUNCTION문입니다.

```
CREATE FUNCTION FOLD(CLOB(100K),INT)
  RETURNS CLOB(100K)
  FENCED
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  LANGUAGE C
  NULL CALL
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME 'udf!fold' ;

CREATE FUNCTION FINDV(VARCHAR(500))
  RETURNS INTEGER
  NOT FENCED
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  LANGUAGE C
  NOT NULL CALL
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME 'udf!findvwl' ;
```

위의 CREATE FUNCTION문은 UNIX 플랫폼을 위한 것입니다. 다른 플랫폼에서는 위 명령문의 EXTERNAL NAME절에 지정된 값을 수정해야 할 것입니다. DB2와 함께 제공된 calludf.sqc 예제 프로그램에서 위의 CREATE FUNCTION 문을 찾을 수 있을 것입니다.

이 CREATE문을 참조하면서 다음에 주의깊게 살펴 보십시오.

- 함수의 스키마 이름은 기본값으로 명령문의 권한 부여 ID입니다.
- FOLD는 FENCED로 정의되었는데, 이는 이것이 완전히 오류가 없는지 확신할 수 없기 때문이며, 그러나 FINDV는 NOT FENCED입니다.
- FOLD에 대하여 NULL CALL을 코딩했는데, 이것은 입력 인수 중 하나가 널(NULL)이라고 하더라도 fold()가 호출될 것이며, 이것은 이 함수가 코딩된 방식과도 일치합니다. FINDV는 NOT NULL CALL로 코딩되었는데, 이것은 코드와 일치합니다.
- 이들 모두 기본값으로 ALLOW PARALLELISM입니다.

이제 다음 명령문을 성공적으로 수행할 수 있습니다.

```
SELECT SUBSTR(DESCR,1,30), SUBSTR(FOLD(DESCR,6),1,30) FROM TEST
```

이 명령문의 CLP 출력은 다음과 같습니다.

```
1                              2
------------------------------ ------------------------------
The only part of the body capa ly part of the body capable of
The seat of the emotions?      at of the emotions?The se
That bendy place in mid-arm.   endy place in mid-arm.That b
-                              INVALID INPUT
Unknown.                       n.Unknow
  5개의 레코드가 선택되었습니다.
```

선택된 CLOB 값이 더 멋지게 표시되도록 하려면 SUBSTR 내장 함수 사용을 고려해 보십시오. 이것은 출력이 접히는 방식을 보여줍니다. 즉 첫번째 행보다 짧은 CLOB 값을 갖는 두 번째, 세 번째 및 다섯 번째 행에서 가장 잘 보이므로, 이렇게 접기는 SUBSTR의 사용으로 더욱 명확해집니다. 또한 입력 텍스트 문자열(컬럼 DESCR)이 널(NULL)인 경우, INVALID INPUT 문자열이 FOLD UDF에 의

해 리턴되는 방식을 보여줍니다(네 번째 행). 또한 이 SELECT에서는 함수 참조의 단순한 중첩을 보여주는데, FOLD에 대한 참조는 SUBSTR 함수 참조의 인수 내에 있습니다.

다음과 같은 명령문을 수행하는 경우,

```
SELECT PART, FINDV(PART) FROM TEST
```

CLP 출력은 다음과 같습니다.

```
PART  2
----- -----------
brain           3
heart           2
elbow           1
-               -
SQL0443N  "SLICK.FINDV" 루틴(특정 이름은 "SQL950424135144750")이
진단 텍스트 "findvwl: No Vowel"와(과) 함께 오류 SQLSTATE를 리턴했습니다.
SQLSTATE=38999
```

이 예에서는 38999 SQLSTATE 값과 findvwl()에 의해 리턴되는 오류 메시지 토큰이 처리되는 방식을 보여줍니다. 메시지 SQL0443N은 이러한 정보를 사용자에게 리턴합니다. 다섯 번째 행의 PART 컬럼에는 모음이 들어 있지 않으며, 이것은 UDF에서 오류를 트리거하는 조건입니다.

이 예에서 인수 승격에 주목하십시오. PART 컬럼은 CHAR(5)이고, FINDV로 전달되기 위해 VARCHAR로 승격됩니다.

마지막으로 DB2가 FINDV에 대한 CREATE문에서 NOT NULL CALL 스펙의 결과로서 네 번째 행에 대하여 FINDV로부터 널(NULL) 출력을 생성하는 방식에 주목하십시오.

다음 명령문은

```
SELECT SUBSTR(DESCR,1,25), FINDV(CAST (DESCR AS VARCHAR(60) ) )
FROM TEST
```

CLP에서 실행될 때 다음과 같은 출력을 생성합니다.

```
1                          2
------------------------- -----------
The only part of the body          3
The seat of the emotions?          3
That bendy place in mid-a          3
-                                  -
Unknown.                           1
  5개의 레코드가 선택되었습니다.
```

이 SELECT문에서는 FINDV가 VARCHAR 입력 인수에 대한 작업을 수행하는 것을 보여줍니다. 이를 위해 컬럼 DESCR을 VARCHAR로 유형변환하는 방식을 주의깊게 살펴 보십시오. 유형변환을 하지 않고는 CLOB에서 FINDV를 사용할 수 없는데, 이는 CLOB가 VARCHAR로 승격되지 않기 때문입니다. 여기에서도 내장 SUBSTR 함수가 사용되어 DESCR 컬럼 값이 더 보기좋게 표시되도록 합니다.

여기에서도 NOT NULL CALL로 인해 네번째 행에서 FINDV로부터의 널(NULL) 결과가 생성됩니다.

예: 계수기(Counter)

사용자의 SELECT문에서 단순히 행에 번호를 붙이고자 한다고 가정하십시오. 사용자는 계수기를 하나씩 증가시키고 리턴하는 UDF를 작성합니다. 이 UDF는 스크래치 패드를 사용합니다.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqludf.h>
#include <sqlca.h>
#include <sqlda.h>

/* structure scr defines the passed scratchpad for the function "ctr" */
struct scr {
   sqlint32 len;
   sqlint32 countr;
   char not_used[96];
} ;

 /*********************************************************************
*  function ctr: increments and reports the value from the scratchpad.
*
*         This function does not use the constructs defined in the
*         "sqludf.h" header file.
*
```

```
*     input:  NONE
*     output: INTEGER out      the value from the scratchpad
*******************************************************************/
#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN ctr (
   sqlint32 *out,                           /* output answer (counter) */
   short *outnull,                      /* output NULL indicator */
   char *sqlstate,                      /* SQL STATE */
   char *funcname,                      /* function name */
   char *specname,                      /* specific function name */
   char *mesgtext,                      /* message text insert */
   struct scr *scratchptr) {            /* scratch pad */

   *out = ++scratchptr->countr;      /* increment counter & copy out */
   *outnull = 0;
}
/* end of UDF : ctr */
```

이 UDF의 경우 다음에 주의하십시오.

- 여기에는 `sqludf.h`가 포함되지 않습니다. 그러나 여기에는 SQL_API_FN의 정의를 위해 `sqlsystm.h`가 포함됩니다.
- 입력 SQL 인수가 정의되어 있지 않지만, 값을 리턴합니다.
- 이것은 네 개의 표준 후미 인수들, 즉 *SQL-state*, *function-name*, *specific-name* 및 *message-text* 뒤에 스크래치 패드 입력 인수를 추가합니다.
- 여기에는 전달되는 스크래치 패드를 맵핑하기 위한 구조 정의가 포함됩니다.

다음은 이 UDF에 대한 CREATE FUNCTION문입니다.

```
CREATE FUNCTION COUNTER()
  RETURNS INT
  SCRATCHPAD
  NOT FENCED
  NOT DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME 'udf!ctr'
  DISALLOW PARALLELISM;
```

이것은 UDF의 AIX 버전에 대한 것입니다. 다른 플랫폼의 경우 EXTERNAL NAME절에 지정된 값을 수정해야 할 것입니다.

이 명령문을 참조하면서, 다음에 주목하십시오.

- 입력 매개변수가 정의되어 있지 않습니다. 이것은 코드와 일치합니다.
- SCRATCHPAD가 코딩되어, DB2가 스크래치 패드 인수를 할당하고, 적절히 초기화하고 전달하도록 합니다.
- 사용자는 프로그램에 오류가 없음을 확실하게 확신하므로 NOT FENCED로 정의했습니다.
- 또한 NOT DETERMINISTIC으로 지정한 것은 SQL 입력 인수 이상에 달려 있기 때문입니다(이 경우에는 없음).
- UDF가 올바르게 기능하는 것은 단일 스크래치 패드에 달려 있으므로, 사용자가 올바르게 DISALLOW PARALLELISM을 지정하였습니다.

이제 다음 명령문을 성공적으로 수행할 수 있습니다.

```
SELECT INT1, COUNTER(), INT1/COUNTER() FROM TEST
```

CLP를 수행할 때, 다음과 같은 출력이 생성됩니다.

```
INT1        2           3
----------- ----------- -----------
         16           1          16
          8           2           4
          4           3           1
          2           4           0
         97           5          19
5개의 레코드가 선택되었습니다.
```

두번째 컬럼에 올바른 COUNTER() 출력이 표시되었습니다. 세번째 컬럼에서는 SELECT문의 COUNTER()에 대한 두 개의 별도의 참조가 각각의 스크래치 패드를 가지고 있음을 보여줍니다. 각각의 스크래치 패드를 가지고 있지 않다면, 두번째 컬럼의 출력은 올바르게 정렬된 1 2 3 4 5 대신 1 3 5 7 9가 될 것입니다.

예: 날씨 테이블 함수

다음은 프로그래밍 예 tblsrv.c에서 DB2에 의해 제공되는 예제 테이블 함수 tfweather_u로서, 미국 내 여러 도시의 날씨 정보를 리턴합니다. 이들 도시에 대한 날씨 정보는 예제 프로그램에 포함되어 있지만, 예제 프로그램 내 주석에서 알 수 있듯이 외부 파일로부터 읽어 들일 수도 있습니다. 이 데이터에는 도시 이름이 나오고 그 다음에 날씨 정보가 뒤따릅니다. 이 패턴은 다른 도시의 경우에도 반복

됩니다. 이 테이블 함수를 호출하고 tfweather_u 테이블 함수를 사용하여 검색된 날씨 데이터를 인쇄하는, DB2와 함께 제공되는 클라이언트 응용프로그램 tblcli.sqc가 있음에 주의하십시오.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sql.h>
#include <sqludf.h> /* for use in compiling User Defined Function */

#define   SQL_NOTNULL   0   /* Nulls Allowed - Value is not Null */
#define   SQL_ISNULL   -1   /* Nulls Allowed - Value is Null */

/* Short and long city name structure */
typedef struct {
  char * city_short ;
  char * city_long ;
} city_area ;

/* Scratchpad data */
/* Preserve information from one function call to the next call */
typedef struct {
  /* FILE * file_ptr; if you use weather data text file */
  int file_pos ;  /* if you use a weather data buffer */
} scratch_area ;

/* Field descriptor structure */
typedef struct {
  char fld_field[31] ;                     /* Field data */
  int  fld_ind ;            /* Field null indicator data */
  int  fld_type ;                          /* Field type */
  int  fld_length ;  /* Field length in the weather data */
  int  fld_offset ;  /* Field offset in the weather data */
} fld_desc ;

/* Short and long city name data */
city_area cities[] = {
  { "alb", "Albany, NY"                  },
  { "atl", "Atlanta, GA"                 },
  .
  .
  .
  { "wbc", "Washington DC, DC"           },
  /* You may want to add more cities here */

  /* Do not forget a null termination */
  { ( char * ) 0, ( char * ) 0           }
} ;

/* Field descriptor data */
fld_desc fields[] = {
  { "", SQL_ISNULL, SQL_TYP_VARCHAR, 30,  0 }, /* city          */
  { "", SQL_ISNULL, SQL_TYP_INTEGER,  3,  2 }, /* temp_in_f     */
  { "", SQL_ISNULL, SQL_TYP_INTEGER,  3,  7 }, /* humidity      */
  { "", SQL_ISNULL, SQL_TYP_VARCHAR,  5, 13 }, /* wind          */
  { "", SQL_ISNULL, SQL_TYP_INTEGER,  3, 19 }, /* wind_velocity */
  { "", SQL_ISNULL, SQL_TYP_FLOAT,    5, 24 }, /* barometer     */
  { "", SQL_ISNULL, SQL_TYP_VARCHAR, 25, 30 }, /* forecast      */
```

```
  /* You may want to add more fields here */

  /* Do not forget a null termination */
  { ( char ) 0, 0, 0, 0, 0 }
} ;

/* Following is the weather data buffer for this example. You */
/* may want to keep the weather data in a separate text file. */
/* Uncomment the following fopen() statement.  Note that you  */
/* have to specify the full path name for this file.          */
char * weather_data[] = {
   "alb.forecast",
   "   34   28%    wnw   3  30.53 clear",
   "atl.forecast",
   "   46   89%   east  11  30.03 fog",
   .
   .
   .
   "wbc.forecast",
   "   38   96%    ene  16  30.31 light rain",
   /* You may want to add more weather data here */

   /* Do not forget a null termination */
   ( char * ) 0
} ;

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* Find a full city name using a short name */
int get_name( char * short_name, char * long_name ) {

    int name_pos = 0 ;

    while ( cities[name_pos].city_short != ( char * ) 0 ) {
       if (strcmp(short_name, cities[name_pos].city_short) == 0) {
          strcpy( long_name, cities[name_pos].city_long ) ;
          /* A full city name found */
   return( 0 ) ;
       }
       name_pos++ ;
    }
    /* Could not find such city in the city data */
    strcpy( long_name, "Unknown City" ) ;
    return( -1 ) ;

}

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* Clean all field data and field null indicator data */
int clean_fields( int field_pos ) {

    while ( fields[field_pos].fld_length != 0 ) {
       memset( fields[field_pos].fld_field, '\0', 31 ) ;
       fields[field_pos].fld_ind = SQL_ISNULL ;
       field_pos++ ;
```

```
    }
   return( 0 ) ;

}

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* Fills all field data and field null indicator data ... */
/* ... from text weather data */
int get_value( char * value, int field_pos ) {

    fld_desc * field ;
    char field_buf[31] ;
    double * double_ptr ;
    int * int_ptr, buf_pos ;

    while ( fields[field_pos].fld_length != 0 ) {
       field = &fields[field_pos] ;
       memset( field_buf, '\0', 31 ) ;
       memcpy( field_buf,
               ( value + field->fld_offset ),
               field->fld_length ) ;
       buf_pos = field->fld_length ;
       while ( ( buf_pos > 0 ) &&
               ( field_buf[buf_pos] == ' ' ) )
          field_buf[buf_pos--] = '\0' ;
       buf_pos = 0 ;
       while ( ( buf_pos < field->fld_length ) &&
               ( field_buf[buf_pos] == ' ' ) )
          buf_pos++ ;
       if ( strlen( ( char * ) ( field_buf + buf_pos ) ) > 0 ||
            strcmp( ( char * ) ( field_buf + buf_pos ), "n/a") != 0 ) {
          field->fld_ind = SQL_NOTNULL ;

          /* Text to SQL type conversion */
          switch( field->fld_type ) {
            case SQL_TYP_VARCHAR:
                 strcpy( field->fld_field,
                         ( char * ) ( field_buf + buf_pos ) ) ;
                 break ;
            case SQL_TYP_INTEGER:
                 int_ptr = ( int * ) field->fld_field ;
                 *int_ptr = atoi( ( char * ) ( field_buf + buf_pos ) ) ;
                 break ;
            case SQL_TYP_FLOAT:
                 double_ptr = ( double * ) field->fld_field ;
                 *double_ptr = atof( ( char * ) ( field_buf + buf_pos ) ) ;
                 break ;
            /* You may want to add more text to SQL type conversion here */
          }

       }
       field_pos++ ;
    }
   return( 0 ) ;

}
```

```
#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN weather( /* Return row fields */
              SQLUDF_VARCHAR * city,
              SQLUDF_INTEGER * temp_in_f,
              SQLUDF_INTEGER * humidity,
              SQLUDF_VARCHAR * wind,
              SQLUDF_INTEGER * wind_velocity,
              SQLUDF_DOUBLE  * barometer,
              SQLUDF_VARCHAR * forecast,
              /* You may want to add more fields here */

              /* Return row field null indicators */
              SQLUDF_NULLIND * city_ind,
              SQLUDF_NULLIND * temp_in_f_ind,
              SQLUDF_NULLIND * humidity_ind,
              SQLUDF_NULLIND * wind_ind,
              SQLUDF_NULLIND * wind_velocity_ind,
              SQLUDF_NULLIND * barometer_ind,
              SQLUDF_NULLIND * forecast_ind,
              /* You may want to add more field indicators here */

              /* UDF always-present (trailing) input arguments */
              SQLUDF_TRAIL_ARGS_ALL
            ) {

  scratch_area * save_area ;
  char line_buf[81] ;
  int line_buf_pos ;

  /* SQLUDF_SCRAT is part of SQLUDF_TRAIL_ARGS_ALL */
  /* Preserve information from one function call to the next call */
  save_area = ( scratch_area * ) ( SQLUDF_SCRAT->data ) ;

  /* SQLUDF_CALLT is part of SQLUDF_TRAIL_ARGS_ALL */
  switch( SQLUDF_CALLT ) {

    /* First call UDF: Open table and fetch first row */
    case SQL_TF_OPEN:
         /* If you use a weather data text file specify full path */
         /* save_area->file_ptr = fopen("/sqllib/samples/c/tblsrv.dat",
                                        "r"); */
         save_area->file_pos = 0 ;
         break ;

    /* Normal call UDF: Fetch next row */
    case SQL_TF_FETCH:
         /* If you use a weather data text file */
         /* memset(line_buf, '\0', 81); */
         /* if (fgets(line_buf, 80, save_area->file_ptr) == NULL) { */
         if ( weather_data[save_area->file_pos] == ( char * ) 0 ) {

            /* SQLUDF_STATE is part of SQLUDF_TRAIL_ARGS_ALL */
            strcpy( SQLUDF_STATE, "02000" ) ;

            break ;
         }
         memset( line_buf, '\0', 81 ) ;
         strcpy( line_buf, weather_data[save_area->file_pos] ) ;
```

```
line_buf[3] = '\0' ;

/* Clean all field data and field null indicator data */
clean_fields( 0 ) ;

/* Fills city field null indicator data */
fields[0].fld_ind = SQL_NOTNULL ;

/* Find a full city name using a short name */
/* Fills city field data */
if ( get_name( line_buf, fields[0].fld_field ) == 0 ) {
   save_area->file_pos++ ;
   /* If you use a weather data text file */
   /* memset(line_buf, '\0', 81); */
   /* if (fgets(line_buf, 80, save_area->file_ptr) == NULL) { */
   if ( weather_data[save_area->file_pos] == ( char * ) 0 ) {
      /* SQLUDF_STATE is part of SQLUDF_TRAIL_ARGS_ALL */
      strcpy( SQLUDF_STATE, "02000" ) ;
      break ;
   }
   memset( line_buf, '\0', 81 ) ;
   strcpy( line_buf, weather_data[save_area->file_pos] ) ;
   line_buf_pos = strlen( line_buf ) ;
   while ( line_buf_pos > 0 ) {
      if ( line_buf[line_buf_pos] >= ' ' )
         line_buf_pos = 0 ;
  else
         line_buf[line_buf_pos] = '\0' ;
         line_buf_pos-- ;
      }
   }
}

/* Fills field data and field null indicator data ... */
/* ... for selected city from text weather data */
get_value( line_buf, 1 ) ;  /* Skips city field */

/* Builds return row fields */
strcpy( city, fields[0].fld_field ) ;
memcpy( (void *) temp_in_f,
        fields[1].fld_field,
        sizeof( SQLUDF_INTEGER ) ) ;
memcpy( (void *) humidity,
        fields[2].fld_field,
        sizeof( SQLUDF_INTEGER ) ) ;
strcpy( wind, fields[3].fld_field ) ;
memcpy( (void *) wind_velocity,
        fields[4].fld_field,
        sizeof( SQLUDF_INTEGER ) ) ;
memcpy( (void *) barometer,
        fields[5].fld_field,
        sizeof( SQLUDF_DOUBLE ) ) ;
strcpy( forecast, fields[6].fld_field ) ;

/* Builds return row field null indicators */
memcpy( (void *) city_ind,
        &(fields[0].fld_ind),
        sizeof( SQLUDF_NULLIND ) ) ;
memcpy( (void *) temp_in_f_ind,
        &(fields[1].fld_ind),
```

```
                sizeof( SQLUDF_NULLIND ) ) ;
        memcpy( (void *) humidity_ind,
                &(fields[2].fld_ind),
                sizeof( SQLUDF_NULLIND ) ) ;
        memcpy( (void *) wind_ind,
                &(fields[3].fld_ind),
                sizeof( SQLUDF_NULLIND ) ) ;
        memcpy( (void *) wind_velocity_ind,
                &(fields[4].fld_ind),
                sizeof( SQLUDF_NULLIND ) ) ;
        memcpy( (void *) barometer_ind,
                &(fields[5].fld_ind),
                sizeof( SQLUDF_NULLIND ) ) ;
        memcpy( (void *) forecast_ind,
                &(fields[6].fld_ind),
                sizeof( SQLUDF_NULLIND ) ) ;

        /* Next city weather data */
        save_area->file_pos++ ;

        break ;

   /* Special last call UDF for cleanup (no real args!): Close table */
   case SQL_TF_CLOSE:
        /* If you use a weather data text file */
        /* fclose(save_area->file_ptr); */
        /* save_area->file_ptr = NULL; */
        save_area->file_pos = 0 ;
        break ;

  }

}
```

이 UDF 코드를 참조하면서 다음을 주의깊게 살펴 보십시오.

- 스크래치 패드가 정의되어 있습니다. row 변수가 OPEN 호출시 초기화되고, iptr 배열과 nbr_rows 변수는 OPEN시에 *mystery* 함수에 의해 채워집니다.
- FETCH는 색인으로 row를 사용하여 iptr 배열을 횡단하고, iptr의 현재 요소로부터 관심이 있는 값을 out_c1, out_c2, 및 out_c3 결과 값 포인터에 의해 가리켜진 위치로 이동합니다.
- 마지막으로 CLOSE는 OPEN에 의해 확보되어 스크래치 패드에 앵커(anchor)된 저장영역을 해제합니다.

다음은 이 UDF에 대한 CREATE FUNCTION문입니다.

```
CREATE FUNCTION tfweather_u()
  RETURNS TABLE (CITY VARCHAR(25),
                 TEMP_IN_F INTEGER,
                 HUMIDITY INTEGER,
                 WIND VARCHAR(5),
```

```
                    WIND_VELOCITY INTEGER,
                    BAROMETER FLOAT,
                    FORECAST VARCHAR(25))
      SPECIFIC tfweather_u
      DISALLOW PARALLELISM
      NOT FENCED
  DETERMINISTIC
        NO SQL
  NO EXTERNAL ACTION
        SCRATCHPAD
      NO FINAL CALL
      LANGUAGE C
        PARAMETER STYLE DB2SQL
      EXTERNAL NAME 'tf_dml!weather';
```

위의 CREATE FUNCTION문은 UNIX 버전의 UDF에 대한 것입니다. 다른 플랫폼의 경우 EXTERNAL NAME절에 지정된 값을 수정해야 할 것입니다.

이 명령문을 참조하면서, 다음에 주목하십시오.

- 아무런 입력을 받아들이지 않고 7개의 출력 컬럼을 리턴합니다.
- SCRATCHPAD가 지정되어 DB2는 스크래치 패드 인수를 할당하고, 적절히 초기화하며, 전달합니다.
- NO FINAL CALL이 지정되었습니다.
- 또한 NOT DETERMINISTIC으로 지정한 것은 SQL 입력 인수 이상에 의존하기 때문입니다. 즉, 이것은 mystery 함수에 종속되며, 그 내용은 매 실행에 따라 달라질 것으로 가정합니다.
- DISALLOW PARALLELISM은 테이블 함수에 필요합니다.
- CARDINALITY 100은 DB2 최적화 알고리즘에 제공되는, 예상되는 리턴 행의 수입니다.
- DBINFO는 사용되지 않았고, 이 함수를 참조하는 특정 명령문에 필요한 컬럼만을 리턴하기 위한 최적화가 구현되지 않았습니다.
- NOT NULL CALL이 지정되어, 입력 SQL 인수 중 하나가 NULL이고 이 조건을점검할 필요가 없는 경우, UDF는 호출되지 않습니다.

예: LOB 위치 지정자를 사용하는 함수

이 UDF는 입력 LOB로 위치 지정자를 받아들이고 입력 LOB의 부속 집합인 다른 LOB에 대한 위치 지정자를 리턴합니다. 두번째 입력 값으로 다른 기준이 전달되는데, 이것은 UDF에게 입력 LOB를 정확하게 나누는 방법을 알려줍니다.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sql.h>
#include <sqlca.h>
#include <sqlda.h>
#include <sqludf.h>
#include "util.h"

void SQL_API_FN lob_subsetter(
          udf_locator * lob_input,   /* locator of LOB value to carve up */
          char     * criteria,          /* criteria for carving */
          udf_locator    * lob_output,  /* locator of result LOB value */
          sqlint16 * inp_nul,
          sqlint16 * cri_nul,
          sqlint16 * out_nul,
          char     * sqlstate,
          char     * funcname,
          char     * specname,
          char     * msgtext ) {

       /* local vars */
       short j;              /* local indexing var */
       int   rc;             /* return code variable for API calls */
       sqlint32  input_len;  /* receiver for input LOB length */
       sqlint32  input_pos;  /* current position for scanning input LOB */
       char lob_buf[100];    /* data buffer */
       sqlint32  input_rec;  /* number of bytes read by sqludf_substr */
       sqlint32  output_rec; /* number of bytes written by sqludf_append */

       /*---------------------------------------------
        * UDF Program Logic Starts Here
        *---------------------------------------------
        * What we do is create an output handle, and then
        * loop over the input, 100 bytes at a time.
        * Depending on the "criteria" passed in, we may decide
        * to append the 100 byte input lob segment to the output, or not.
        *---------------------------------------------
        * Create the output locator, right in the return buffer.
        */

       rc = sqludf_create_locator(SQL_TYP_CLOB, &lob_output);
       /* Error and exit if unable to create locator */
       if (rc) {
          memcpy (sqlstate, "38901", 5);
          /* special sqlstate for this condition */
          goto exit;
       }
       /* Find out the size of the input LOB value */
       rc = sqludf_length(lob_input, &input_len) ;
       /* Error and exit if unable to find out length */
```

```
      if (rc) {
         memcpy (sqlstate, "38902", 5);
         /* special sqlstate for this condition */
         goto exit;
      }
      /* Loop to read next 100 bytes, and append to result if it meets
       * the criteria.
       */
      for (input_pos = 0; (input_pos < input_len); input_pos += 100) {
        /* Read the next 100 (or less) bytes of the input LOB value */
        rc = sqludf_substr(lob_input, input_pos, 100,
                           (unsigned char *) lob_buf, &input_rec) ;
        /* Error and exit if unable to read the segment */
        if (rc) {
           memcpy (sqlstate, "38903", 5);
           /* special sqlstate for this condition */
           goto exit;
        }
        /* apply the criteria for appending this segment to result
         * if (...predicate involving buffer and criteria...) {
         * The condition for retaining the segment is TRUE...
         * Write that buffer segment which was last read in
         */
        rc = sqludf_append(lob_output,
                   (unsigned char *) lob_buf, input_rec, &output_rec) ;
        /* Error and exit if unable to read the 100 byte segment */
        if (rc) {
           memcpy (sqlstate, "38904", 5);
           /* special sqlstate for this condition */
           goto exit;
        }
        /* } end if criteria for inclusion met */
      } /* end of for loop, processing 100-byte chunks of input LOB
        * if we fall out of for loop, we are successful, and done.
 */
   *out_nul = 0;
exit: /* used for errors, which will override null-ness of output. */
   return;
}
```

이 UDF 코드를 참조하면서 다음을 주의깊게 살펴 보십시오.

- 여기에는 sqludf_create_locator() 호출에서 사용되는 SQL_TYP_CLOB 유형이 정의되어 있는 sql.h와 udf_locator 유형이 정의되어 있는 sqludf.h가 포함됩니다.
- 첫번째 입력 인수와 함수 출력을 나타내는 세번째 입력 인수는 sqludf_locator에 대한 포인터로 정의되는데, 이들은 AS LOCATOR의 CREATE FUNCTION 스펙을 나타냅니다.
- UDF는 CREATE FUNCTION문에 NOT NULL CALL이 지정되었기 때문에, 입력 인수 중 어느 하나가 널(NULL)인지 테스트는 하지 않았습니다.

- 오류가 발생하면, UDF는 sqlstate를 38*xxx*로 설정하고 종료합니다. 이것은 UDF를 참조하는 명령문의 실행을 중단하기에 충분합니다. 사용자가 선택한 실제 38*xxx* SQLSTATE 값은 DB2에 있어서는 중요하지 않지만, 사용자의 UDF에서 발생할 수 있는 예외 조건을 판별하는 데 사용됩니다.
- 포함 기준은 지정하지 않고 남겨두었지만, 이 경우 이 특정 버퍼의 내용이 테스트를 통과하는지를 판별할 수 있고, 마지막 버퍼가 부분 버퍼일 가능성을 설명할 수 있습니다.
- 추가된 데이터 길이로 input_rec 변수를 사용함으로써, UDF는 부분적인 버퍼 조건을 점검합니다.

다음은 이 UDF에 대한 CREATE FUNCTION문입니다.

```
     CREATE FUNCTION carve(CLOB(50M), VARCHAR(255) )
       RETURNS CLOB(50M)
       NOT NULL CALL
       NOT FENCED
   DETERMINISTIC
         NO SQL
   NO EXTERNAL ACTION
       LANGUAGE C
         PARAMETER STYLE DB2SQL
       EXTERNAL NAME '/u/wilfred/udfs/lobudfs!lob_subsetter' ;
```

이것은 UDF의 AIX 버전에 대한 것입니다. 다른 플랫폼의 경우 EXTERNAL NAME절에 지정된 값을 수정해야 할 것입니다.

이 명령문을 참조하면서, 다음에 주목하십시오.

- NOT NULL CALL이 지정되어, 입력 SQL 인수 중 하나가 NULL이고 이 조건을점검할 필요가 없는 경우, UDF는 호출되지 않습니다.
- 이 함수는 NOT FENCED로 정의되어 있습니다. 이 API는 NOT FENCED에서만 작동함에 주의하십시오. NOT FENCED는 정의자가 데이터베이스에 대하여 CREATE_NOT_FENCED 권한을 가져야 할 것임을 의미하며, 이것은 DBADM 권한에 의해서도 같은 의미를 갖습니다.
- 이 함수는 DETERMINISTC으로 지정되었는데, 이것은 주어진 CLOB 값과 기준 세트로는 결과가 항상 동일할 것임을 의미합니다.

이제 다음 명령문을 성공적으로 수행할 수 있습니다.

```
UPDATE tablex
  SET col_a = 99,
      col_b = carve (:hv_clob, '...criteria...')
  WHERE tablex_key = :hv_key;
```

이 UDF는 호스트 변수 :hv_clob으로 표시되는 CLOB 값을 부속 집합으로 만들고, 호스트 변수 :hv_key의 키 값으로 표시되는 행을 갱신하는 데 사용됩니다.

이 갱신 예제에서, :hv_clob는 응용프로그램에서 CLOB_LOCATOR로 정의됩니다. 이것은 "카브" UDF로 전달되는 위치 지정자와 같은 값이 아닙니다. :hv_clob가 명령문을 수행하는 DB2 엔진 에이전트에 "바인드"될 때, 이것은 CLOB로만 알려집니다. 그리고나서 UDF로 전달될 때, DB2는 이 값에 대한 새로운 위치 지정자를 생성합니다. 이러한 CLOB과 위치 지정자 사이의 변환은 비용이 많이 소모되는 것이 아닙니다. 필요 이상의 메모리 사본이나 입출력을 필요로 하지 않습니다.

예: BASIC으로 작성한 계수기(Counter) OLE 자동 UDF

다음의 예에서는 Microsoft Visual BASIC을 사용하여 계수기 클래스를 구현합니다. 이 클래스에는 호출 횟수를 추적하는 인스턴스 변수 nbrOfInvoke가 있습니다. 클래스 구성자는 이 숫자를 0으로 초기화합니다. Increment 메소드가 nbrOfInvoke를 1씩 증가시키고 현재 상태를 리턴합니다.

```
Description="Example in SQL Reference"
Name="bert"
Class=bcounter; bcounter.cls
ExeName32="bert_app.exe"


VERSION 1.0 CLASS
BEGIN
      SingleUse = -1  'True
END
     Attribute VB_Name = "bcounter"
Attribute VB_Creatable = True
Attribute VB_Exposed = True
Option Explicit
     Dim nbrOfInvoke As Long

     Public Sub increment(output As Long, _
```

```
                                  output_ind As Integer, _
                                  sqlstate As String, _
                                  fname As String, _
                                  fspecname As String, _
                                  msg As String, _
                                  scratchpad() As Byte, _
                                  calltype As Long)

         nbrOfInvoke = nbrOfInvoke + 1

End Sub

     Private Sub Class_Initialize()
       nbrOfInvoke = 0
End Sub

     Private Sub Class_Terminate()

End Sub
```

bcounter 클래스는 OLE 자동 오브젝트로서 구현되고, progId bert.bcounter 아래에 등록됩니다. 자동 서버를 프로세스 내부 프로세스 또는 지역 서버 중 하나로 컴파일할 수 있습니다. 이것은 DB2에 투명합니다. 다음의 CREATE FUNCTION문은 외부 구현으로 increment 메소드를 갖는 UDF bcounter를 등록합니다.

```
CREATE FUNCTION bcounter () RETURNS integer
   EXTERNAL NAME 'bert.bcounter!increment'
    LANGUAGE OLE
    FENCED
    SCRATCHPAD
    FINAL CALL
    NOT DETERMINISTIC
    NULL CALL
    PARAMETER STYLE DB2SQL
    NO SQL
NO EXTERNAL ACTION
    DISALLOW PARALLEL;
```

다음과 같은 조회의 경우,

```
SELECT INT1, BCOUNTER() AS COUNT, INT1/BCOUNTER() AS DIV FROM TEST
```

결과는 이전 예에서와 정확히 똑같습니다.

```
INT1        COUNT       DIV
----------- ----------- -----------
         16           1          16
          8           2           4
          4           3           1
          2           4           0
         97           5          19

  5개의 레코드가 선택되었습니다.
```

예: C++로 작성한 계수기(Counter) OLE 자동 UDF

다음의 예에서는 앞의 BASIC 계수기 클래스를 C++로 구현합니다. 코드의 일부만이 여기에 나와 있으며, 전체 샘플은 /sqllib/samples/ole 디렉토리에 있습니다.

Increment 메소드는 계수기 인터페이스 설명의 일부로 오브젝트 설명 언어에서 설명합니다.

```
interface ICounter : IDispatch
{
    ...
    HRESULT increment([out] long    *out,
                      [out] short   *outnull,
                      [out] BSTR    *sqlstate,
                      [in]  BSTR    *fname,
                      [in]  BSTR    *fspecname,
                      [out] BSTR    *msgtext,
                      [in,out] SAFEARRAY (unsigned char) *spad,
                      [in]  long    *calltype);
    ...
}
```

C++의 COM CCounter 클래스 정의에는 nbrOfInvoke와 함께 Increment 메소드 선언도 포함됩니다.

```
class FAR CCounter : public ICounter
{
        ...
STDMETHODIMP CCounter::increment(long    *out,
                                 short   *outnull,
                                 BSTR    *sqlstate,
                                 BSTR    *fname,
                                 BSTR    *fspecname,
                                 BSTR    *msgtext,
                                 SAFEARRAY **spad,
```

```
                                                     long *calltype );
             long nbrOfInvoke;
             ...
     };
```

메소드의 C++ 구현은 BASIC 코드와 비슷합니다.

```
     STDMETHODIMP CCounter::increment(long    *out,
                                      short   *outnull,
                                      BSTR    *sqlstate,
                                      BSTR    *fname,
                                      BSTR    *fspecname,
                                      BSTR    *msgtext,
                                      SAFEARRAY **spad,
                                      long *calltype)
     {

       nbrOfInvoke = nbrOfInvoke + 1;
       *out = nbrOfInvoke;

       return NOERROR;
     };
```

위의 예에서 sqlstate 및 msgtext는 BSTR* 유형의 [out] 매개변수입니다. 즉 DB2는 NULL에 대한 포인터를 UDF로 전달합니다. 이들 매개변수 값을 리턴하기 위해 UDF는 문자열을 할당하고 이를 DB2로 리턴하며(예를 들어 *sqlstate = SysAllocString (L"01H00")), DB2는 메모리를 해제합니다. fname와 fspecname 매개변수는 [in] 매개변수입니다. DB2는 메모리를 할당하고 UDF에 의해 읽힌 값을 전달하고, DB2는 메모리를 해제합니다.

CCounter 클래스의 factory 클래스는 계수기 오브젝트를 작성합니다. factory 클래스를 단일 사용 또는 복수 사용 오브젝트로 등록할 수 있습니다(이 예에서는 나오지 않음).

```
     STDMETHODIMP CCounterCF::CreateInstance(IUnknown FAR* punkOuter,
                                             REFIID riid,
                                             void FAR* FAR* ppv)
     {

       CCounter *pObj;
             ...
```

```
        // create a new counter object
        pObj = new CCounter;
              ...
      };
```

CCounter 클래스는 지역 서버로 구현되었으며, 이것은 progId bert.ccounter 아래에 등록되었습니다. 다음의 CREATE FUNCTION문은 외부 구현으로 Increment 메소드를 갖는 UDF ccounter를 등록합니다.

```
    CREATE FUNCTION ccounter () RETURNS integer
      EXTERNAL NAME 'bert.ccounter!increment'
         LANGUAGE OLE
         FENCED
         SCRATCHPAD
         FINAL CALL
         NOT DETERMINISTIC
         NULL CALL
         PARAMETER STYLE DB2SQL
         NO SQL
   NO EXTERNAL ACTION
         DISALLOW PARALLEL;
```

다음 조회를 처리하는 동안 DB2는 클래스 CCounter의 서로 다른 두 개의 인스턴스를 작성합니다. 조회에서 각 UDF 참조에 대하여 하나의 인스턴스가 작성됩니다. ccounter UDF 등록에 스크래치 패드 옵션이 지정되므로, 두 개의 인스턴스가 전체 조회에 재사용됩니다.

```
SELECT INT1, CCOUNTER() AS COUNT, INT1/CCOUNTER() AS DIV FROM TEST
```

결과는 이전 예에서와 정확히 똑같습니다.

```
INT1        COUNT       DIV
----------- ----------- -----------
         16           1          16
          8           2           4
          4           3           1
          2           4           0
         97           5          19

  5개의 레코드가 선택되었습니다.
```

예: BASIC으로 작성한 메일 OLE 자동 테이블 함수

다음의 예에서는 Microsoft Exchange의 메시지 헤더 정보와 부분적인 메시지 텍스트를 검색하기 위해 공용 메소드 list를 공개하는 Microsoft Visual BASIC

을 사용하는 클래스를 구현합니다. 메소드 구현시 MAPI(메시징 API)에 대한 OLE 자동 인터페이스를 제공하는 OLE 메시징을 사용합니다.

```
Description="Mail OLE Automation Table Function"
Module=MainModule; MainModule.bas
Class=Header; Header.cls
ExeName32="tfmapi.dll"
Name="TFMAIL"

VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
END
Attribute VB_Name = "Header"
Attribute VB_Creatable = True
Attribute VB_Exposed = True
Option Explicit

Dim MySession As Object
Dim MyMsgColl As Object
Dim MyMsg As Object
Dim CurrentSender As Object
Dim name As Variant
Const SQL_TF_OPEN = -1
Const SQL_TF_CLOSE = 1
Const SQL_TF_FETCH = 0


Public Sub List(timereceived As Date, subject As String, size As Long, _
                text As String, ind1 As Integer, ind2 As Integer, _
                ind3 As Integer, ind4 As Integer, sqlstate As String, _
                fname As String, fspecname As String, msg As String, _
                scratchpad() As Byte, calltype As Long)


    If (calltype = SQL_TF_OPEN) Then

       Set MySession = CreateObject("MAPI.Session")

       MySession.Logon ProfileName:="Profile1"
       Set MyMsgColl = MySession.Inbox.Messages

       Set MyMsg = MyMsgColl.GetFirst

    ElseIf (calltype = SQL_TF_CLOSE) Then

       MySession.Logoff
       Set MySession = Nothing

      Else

      If (MyMsg Is Nothing) Then

         sqlstate = "02000"
```

```
    Else

        timereceived = MyMsg.timereceived
        subject = Left(MyMsg.subject, 15)
        size = MyMsg.size
        text = Left(MyMsg.text, 30)

        Set MyMsg = MyMsgColl.GetNext

    End If
    End If
End Sub
```

테이블 함수 OPEN 호출에서 CreateObject문은 메일 세션을 작성하고, logon 메소드는 메일 시스템으로 로그온합니다(사용자 이름과 암호 발행은 무시됨). 메일 수신 상자의 메시지 콜렉션은 첫번째 메시지를 검색하는 데 사용됩니다. FETCH 호출에서 메시지 헤더 정보와 현재 메시지의 처음 30자가 테이블 함수 출력 매개변수에 지정됩니다. 아무런 메시지도 남아 있지 않은 경우, SQLSTATE 02000이 리턴됩니다. CLOSE 호출에서, 이 예는 로그 오프하고 세션 오브젝트에 아무것도 설정하지 않는데, 이것은 다른 변수가 이것을 참조하지 않는 한 이전에 참조된 오브젝트와 연관된 시스템 및 메모리 자원을 모두 릴리스하는 것입니다.

다음은 이 UDF에 대한 CREATE FUNCTION문입니다.

```
CREATE FUNCTION MAIL()
    RETURNS TABLE (TIMERECIEVED DATE,
                   SUBJECT VARCHAR(15),
                   SIZE INTEGER,
                   TEXT VARCHAR(30))
    EXTERNAL NAME 'tfmail.header!list'
    LANGUAGE OLE
    PARAMETER STYLE DB2SQL
    NOT DETERMINISTIC
    FENCED
    NULL CALL
    SCRATCHPAD
    FINAL CALL
    NO SQL
    EXTERNAL ACTION
    DISALLOW PARALLEL;
```

다음은 샘플 조회입니다.

```
SELECT * FROM TABLE (MAIL()) AS M

TIMERECEIVED SUBJECT         SIZE        TEXT
------------ --------------- ----------- ------------------------------
```

```
01/18/1997   Welcome!                3277 Welcome to Windows Messaging!
01/18/1997   Invoice                 1382 Please process this invoice. T
01/19/1997   Congratulations         1394 Congratulations to the purchas

  3개의 레코드가 선택되었습니다.
```

UDF 디버깅

데이터베이스에 손상을 입히지 않을 수 있는 환경에서 UDF를 디버그하는 것이 중요합니다. 사용자의 UDF가 절대적으로 오류가 없음을 확신할 때까지는 테스트 데이터베이스 인스턴스에 대하여 테스트를 실행해야 합니다. FENCED와 NOT FENCED UDF에 대해 양쪽 유형이 제대로 작성되지 않으면 DB2의 오작동을 일으킬 수 있습니다. UDF를 FENCED로 정의하면 NOT FENCED보다 무결성 및 보안 노출에 있어서 더 많은 보호를 제공하지만 이를 보장할 수는 없습니다. 검토 및 테스트가 포함된 좋은 코딩 습관은 그 어느 경우에도 적용되어야 합니다.

DB2는 저장영역을 잘못 수정하는 특정 유형의 제한 조치를 점검합니다. 예를 들어 UDF가 스크래치 패드나 결과 버퍼로 너무 많은 문자들을 이동시킨 경우가 이에 해당됩니다. 이렇게 잘못된 기능을 발견하게 되면 DB2는 오류, SQLCODE -450(SQLSTATE 39501)을 리턴합니다. DB2 역시 UDF에 비정상 종료가 발생하여 SQLCODE -430(SQLSTATE 38503)을 리턴하는 경우 또는 UDF 처리를 사용자가 인터럽트 하여 SQLCODE -431(SQLSTATE 38504)을 리턴하는 경우 치명적인 장애가 되지 않도록 설계되어 있습니다.

FENCED UDF에서도 리턴 값 버퍼가 대량으로 겹쳐쓰여지면 UDF와 DB2가 비정상적으로 종료됩니다. 바이트를 리턴 값 버퍼로 이동시키는 모든 UDF를 설계, 코딩 그리고 검토할 때 특히 주의하십시오. 예를 들어, 바이트를 이동시키기 전에 얼마만큼의 바이트를 이동시켜야 하는지 계산하는 UDF에 대해서는 조심하십시오. C에서 memcpy는 이 함수에 대해 종종 사용됩니다. 바이트를 리턴 값 버퍼로 이동시키는 UDF에 대해 경계 경우(특히 짧거나 긴 값)를 자세하 살펴 보십시오.

보안 및 데이터베이스 무결성을 이유로, 사용자의 UDF가 일단 디버그되고 DB2에 정의된 경우, UDF의 내용을 보호하는 것이 중요합니다. 이것은 사용자의 UDF가 NOT FENCED로 정의되어 있는 경우 더욱 그렇습니다. 실수든 의도적이든, 사용자 자신을 포함하여 누군가가 디버그되지 않은 코드로 작동가능한 UDF를 겹쳐쓰

기 하였을 경우, 이것이 NOT FENCED로 정의되어 있다면 데이터베이스를 손상시킬 가능성이 충분하거나 보안 상태를 위협할 수 있습니다.

안타깝게도 소스 레벨 디버거를 UDF에서 쉽게 수행시킬 방법은 없습니다. 그 이유는 다음과 같습니다.

- UDF가 저장영역에 있어 사용가능한 시점에 디버거를 시작할 타이밍을 포착하기가 어렵습니다.
- UDF는 특별 사용자 ID를 사용하여 데이터베이스 프로세스에서 실행되고 사용자는 이 프로세스에 접속되지 못합니다.

UDF는 보통 `stdout`이 의미가 없는 백그라운드 프로세스에서 수행되기 때문에, `printf()`와 같은 유용한 디버깅 도구가 UDF에서는 작동하지 않습니다. `printf()`의 대안으로 UDF에 파일 출력 논리를 지정하고, 디버깅을 목적으로 표시 데이터와 제어 정보를 파일에 기록할 수 있습니다.

UDF를 디버그하는 또 다른 기법으로 데이터베이스 환경 밖에서 UDF를 호출하는 드라이버 프로그램을 작성할 수 있습니다. 이 기법을 사용하여 사용자는 많은 종류의 경계 값 및 오류 값을 입력 인수로 지정하여 UDF를 호출함으로써, UDF 오류를 유도할 수 있을 것입니다. 이러한 환경에서는 `printf()`나 소스 레벨 디버거를 사용하는 것이 문제가 되지 않습니다.

제16장 활동중인 DBMS에서 트리거 사용

트리거를 사용하는 이유. 537
트리거의 이점 539
트리거 개요. 539
트리거 이벤트 540
영향을 받는 행 집합. 541
트리거 수준. 542
트리거 활성화 시간 543
전이 변수 544
전이 테이블. 546
트리거 조치. 548
트리거 조치 조건 548
트리거 SQL문 549
SQL 트리거 명령문 내의 함수 549
트리거 연쇄. 551
참조 제한조건과의 상호작용 551
복수 트리거 정렬 551
트리거, 제한조건, UDT, UDF 및 LOB간의
시너지 553
정보 추출 553
테이블에 대한 조작 금지 554
비지니스 규칙 정의 554
조치 정의 555

트리거를 사용하는 이유

수동적인 시스템에서 활동적인 시스템으로 사용자의 데이터베이스 관리 프로그램을 변경하려면, 트리거 함수에 내장되어 있는 기능을 사용하십시오. *트리거*는 지정된 기본 테이블에서 갱신 조작에 의해 활성화되거나 *트리거되는* 조치 세트를 정의합니다. 이러한 조치는 데이터베이스에 대한 다른 변경을 발생시키고, DB2 외부에서의 조작을 수행하며(예를 들어, 전자우편을 보내거나 파일에 기록을 작성하는 등), 갱신 조작이 발생하지 않도록 하는 예외를 발생시키기도 합니다.

트리거를 사용하여 사용자는 비지니스 규칙과 같은 일반적인 형태의 무결성을 지원합니다. 예를 들어 사용자의 비지니스에서 고객의 신용 한도를 초과하는 주문은 거절하려고 합니다. 트리거는 이러한 제약을 지정할 때 사용할 수 있습니다. 일반적으로 트리거는 *과도적인* 비지니스 규칙을 캡쳐하기 위한 강력한 메카니즘입니다. 과도적인 비지니스 규칙은 서로 다른 상태의 데이터들이 관련되는 규칙입니다.

예를 들어 월급이 10 퍼센트 이상 증가할 수 없다고 가정하십시오. 이 규칙을 점검하기 위해 증가 전과 증가 후의 월급이 비교되어야 합니다. 둘 이상의 상태를 갖는 데이터와 관련되지 않는 규칙의 경우, 점검 및 참조 무결성 제한조건이 더 적

절할 것입니다. 자세한 내용은 *SQL 참조서* 책을 참조하십시오. 점검 및 참조 제한조건은 그것의 선언적 구문으로 인해, 과도적이지 않은 제한조건에 그 사용이 권장됩니다.

자동으로 요약 데이터를 갱신하는 것과 같은 타스크에도 트리거를 사용할 수 있습니다. 이러한 조치를 데이터베이스의 일부로 설정하고, 이것이 자동적으로 발생하도록 함으로써, 트리거는 데이터베이스의 무결성을 향상시킵니다. 예를 들어, 회사에서 관리하는 직원 수를 자동으로 추적하고자 합니다.

```
Tables: EMPLOYEE (as in Sample Tables)
  COMPANY_STATS (NBEMP, NBPRODUCT, REVENUE)
```

두 가지 트리거를 정의할 수 있습니다.

- 새로운 사람이 고용될 때마다, 즉 새로운 행이 EMPLOYEE 테이블에 삽입될 때마다 직원 수를 증가시키는 트리거.

```
CREATE TRIGGER NEW_HIRED
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

- 직원이 회사를 떠날 때마다, 즉 행이 EMPLOYEE 테이블에서 삭제될 때마다 직원 수를 감소시키는 트리거.

```
CREATE TRIGGER FORMER_EMP
  AFTER DELETE ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  UPDATE COMPANY_STATS SET NBEMP = NBEMP - 1
```

특히 다음의 경우에 트리거를 사용할 수 있습니다.

- SIGNAL SQLSTATE SQL문과 내장 RAISE_ERROR 함수를 사용하여 입력 데이터를 검증하거나, 유효하지 않은 데이터가 발견되었을 때 오류가 발생했음을 나타내는 SQLSTATE를 리턴하도록 UDF를 호출하는 경우. 비 과도적 데이터의 검증이 보통 점검과 참조 제한조건에 의해 더 잘 처리됨에 주의하십시오. 반대로 트리거는 과도적 데이터의 검증, 즉 갱신 조작 이전과 이후 값의 비교를 요하는 검증에 적절합니다.
- 새로 삽입된 행에 자동으로 값을 생성하는 경우(이것을 *대리 함수(surrogate function)*라고 함). 즉, 다른 테이블의 값이나 행의 다른 값에 근거하는 사용자 정의 기본값을 구현하려는 경우.

- 상호 참조를 목적으로 다른 테이블로부터 읽는 경우.
- 감사 추적을 목적으로 다른 테이블에 기록하는 경우.
- *경보*를 지원하는 경우(예를 들어 전자 메일 메시지 등을 통해)

트리거의 이점

사용자의 데이터베이스 관리 프로그램에서 트리거를 사용하여 다음과 같은 결과를 얻을 수 있습니다.

- **더욱 빠른 응용프로그램 개발.**

  트리거는 관계형 데이터베이스에 저장되어 있으므로, 트리거가 수행하는 조치는 각 응용프로그램에서 코딩할 필요가 없습니다.

- **비지니스 규칙의 전역 적용**

  트리거가 한 번만 정의되면, 테이블을 변경하는 어떠한 응용프로그램에서도 사용할 수 있습니다.

- **더욱 용이한 유지보수**

  비지니스 규정이 변경되면, 모든 응용프로그램 대신 해당되는 트리거만을 변경하면 됩니다.

트리거 개요

트리거를 작성할 때 이것을 테이블과 연관시킵니다. 이 테이블을 트리거의 *주제 테이블*이라고 합니다. *갱신 조작*이라는 용어는 주제 테이블의 상태에 발생한 변경을 말합니다. 갱신 조작은 다음에 의해 시작됩니다.

- INSERT문
- UPDATE문 또는 UPDATE를 수행하는 참조 제한조건
- DELETE문 또는 DELETE를 수행하는 참조 제한조건

각 트리거를 이들 세 가지 유형의 갱신 조작 중 하나와 연관시켜야 합니다. 이러한 연관을 해당 특정 트리거에 대한 *트리거 이벤트*라고 합니다.

또한 트리거 이벤트가 발생할 때 트리거가 수행하는 *트리거 조치*라고 하는 조치를 정의해야 합니다. 트리거 조치는 데이터베이스 관리 프로그램이 트리거 이벤트를 수행하기 전 또는 후에 실행될 수 있는 하나 이상의 SQL문으로 구성됩니다. 일단

트리거 이벤트가 발생하면, 데이터베이스 관리 프로그램은 갱신 조작이 영향을 미칠 주제 테이블의 행 집합을 판별하고, 트리거를 실행합니다.

트리거를 작성할 때, 다음과 같은 속성과 동작을 선언합니다.

- 트리거의 이름.
- 주제 테이블의 이름.
- 트리거 활성화 시간(갱신 조작이 실행되기 전 또는 후).
- 트리거 이벤트(INSERT, DELETE 또는 UPDATE).
- 오래된 값 전이 변수.
- 새로운 값 전이 변수.
- 오래된 값 전이 테이블.
- 새로운 값 전이 테이블.
- 수준(FOR EACH STATEMENT 또는 FOR EACH ROW).
- 트리거 조치 조건 및 트리거 SQL문을 포함하는 트리거의 트리거 조치.
- 트리거 이벤트가 UPDATE인 경우, 트리거의 트리거 이벤트에 대한 트리거 컬럼 목록과 트리거 컬럼 목록이 명시적인지 내재된 것인지 여부를 나타낸 것.
- 트리거 작성 시간소인
- 현재 함수 경로

CREATE TRIGGER문에 대한 자세한 정보는 *SQL 참조서* 책을 참조하십시오.

트리거 이벤트

모든 트리거는 이벤트와 연관됩니다. 트리거는 해당 이벤트가 데이터베이스에서 발생하면 활성화됩니다. 이 트리거 이벤트는 참조 제한조건 조치에 의해 발생된 것들을 포함하여 UPDATE, INSERT 또는 DELETE 중 하나가 주제 테이블에서 수행될 때 발생합니다. 예를 들면 다음과 같습니다.

```
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

위의 명령문은 `new_hire` 트리거를 정의하는데, 이것은 `employee` 테이블에 삽입 조작을 수행할 때 활성화됩니다.

사용자는 모든 트리거 이벤트와 모든 트리거를 정확히 하나의 주제 테이블 및 하나의 갱신 조작에 연관시킵니다. 갱신 조작은 다음과 같습니다.

삽입 조작

삽입 조작은 INSERT문에 의해서만 생성될 수 있습니다. 그러므로 데이터가 LOAD 명령과 같이 INSERT를 사용하지 않는 유틸리티를 사용하여 로드될 때는 활성화되지 않습니다.

갱신 조작

갱신 조작은 UPDATE문이나 ON DELETE SET NULL의 참조 제한조건 규칙의 결과 발생할 수 있습니다.

삭제 조작

삭제 조작은 DELETE문이나 ON DELETE CASCADE의 참조 제한조건 규칙의 결과로 발생할 수 있습니다.

트리거 이벤트가 갱신 조작인 경우, 이벤트는 주제 테이블의 특정 컬럼과 연관될 수 있습니다. 이 경우, 트리거는 갱신 조작이 지정된 컬럼 중 임의의 컬럼을 갱신하려고 시도할 때에만 활성화됩니다. 이것으로 트리거를 활성화시키는 이벤트가 더욱 다듬어질 수 있습니다. 예를 들어 다음과 같은 트리거 REORDER는 사용자가 PARTS 테이블의 ON_HAND 또는 MAX_STOCKED 컬럼에 갱신 조작을 수행하는 경우에만 활성화됩니다.

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N_ROW
  FOR EACH ROW MODE DB2SQL
  WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED)
  BEGIN ATOMIC
  VALUES(ISSUE_SHIP_REQUEST(N_ROW.MAX_STOCKED -
                            N_ROW.ON_HAND,
                            N_ROW.PARTNO));
  END
```

영향을 받는 행 집합

트리거 이벤트는 해당 SQL 조작에 의해 영향을 받는 주제 테이블의 행 집합을 정의합니다. 예를 들어 다음과 같은 UPDATE문을 parts 테이블에 수행한다고 가정하십시오.

```
UPDATE PARTS
  SET ON_HAND = ON_HAND + 100
  WHERE  PART_NO > 15000
```

연관된 트리거에 대한 영향을 받는 행 집합에는 *part_no*가 15 000보다 큰 parts 테이블의 행이 모두 들어 있습니다.

트리거 수준

트리거가 활성화되면, 다음과 같은 수준에 따라 수행됩니다.

FOR EACH ROW
: 영향을 받은 행 집합에서 행 수만큼의 횟수로 수행됩니다.

FOR EACH STATEMENT
: 전체 트리거 이벤트에 대하여 한 번 수행됩니다.

영향을 받은 행 집합이 비어 있는 경우, 즉 WHERE절에서 어떠한 행도 조건을 충족시키지 못한 검색된 UPDATE나 DELETE의 경우, FOR EACH ROW 트리거는 수행되지 않습니다. 그러나 FOR EACH STATEMENT 트리거는 한 번 수행됩니다.

예를 들어, 직원 수를 계속 세고 있는 것은 FOR EACH ROW를 사용하여 가능한 것입니다.

```
CREATE TRIGGER NEW_HIRED
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

FOR EACH STATEMENT의 수준을 사용하여 한 번의 갱신으로 동일한 영향을 미칠 수 있습니다.

```
CREATE TRIGGER NEW_HIRED
  AFTER INSERT ON EMPLOYEE
  REFERENCING NEW_TABLE AS NEWEMPS
  FOR EACH STATEMENT MODE DB2SQL
  UPDATE COMPANY_STATS
  SET NBEMP = NBEMP + (SELECT COUNT(*) FROM NEWEMPS)
```

주: FOR EACH STATEMENT의 수준은 『트리거 활성화 시간』에서 논의되는 BEFORE 트리거에 대하여 지원되지 않습니다.

트리거 활성화 시간

*트리거 활성화 시간*은 트리거가 활성화되어야 하는 시간을 지정합니다. 즉, 트리거 이벤트가 실행되는 전(BEFORE) 또는 후(AFTER)입니다. 예를 들어, 다음 트리거의 활성화 시간은 employee에 대한 INSERT 조작 후(AFTER)입니다.

```
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

활성화 시간이 BEFORE인 경우, 트리거 조치는 트리거 이벤트가 실행되기 전 영향을 받은 행 집합의 각 행에 대하여 활성화됩니다. BEFORE 트리거는 FOR EACH ROW 수준이어야 함에 주의하십시오.

활성화 시간이 AFTER인 경우, 트리거 조치는 트리거 수준에 따라, 영향을 받은 행 집합의 각 행에 대하여 또는 명령문에 대하여 활성화됩니다. 이것은 트리거 이벤트가 실행된 후 및 데이터베이스 관리 프로그램이 참조 제한조건의 조치를 포함하여 트리거 이벤트가 영향을 줄 수 있는 모든 조건을 점검한 후에 발생합니다. AFTER 트리거는 FOR EACH ROW 또는 FOR EACH STATEMENT 중 하나의 수준이어야 함에 주의하십시오.

서로 다른 트리거 활성화 시간은 트리거의 목적이 서로 다름을 의미합니다. 기본적으로 BEFORE 트리거는 데이터베이스 관리 시스템의 제한조건 서브시스템의 확장입니다. 그러므로 사용자는 다음을 수행하기 위해 이들을 사용합니다.

- 입력 데이터 검증 수행.
- 새로 삽입된 행 값의 자동 생성.
- 상호 참조를 목적으로 다른 테이블로부터 읽기.

BEFORE 트리거는 트리거 이벤트가 데이터베이스에 적용되기 전에 활성화되므로, 더 깊이 데이터베이스를 수정하는 데 사용되지 않습니다. 결국 이것은 무결성 제한조건이 점검되고 트리거 이벤트에 의해 위반될 수 있다고 하더라도 그 전에 활성화됩니다.

반대로 AFTER 트리거는 특정 이벤트가 발생할 때마다 데이터베이스에서 수행되는 응용프로그램 논리 모듈입니다. 응용프로그램의 일부로, AFTER 트리거는 항상 일관된 상태의 데이터베이스를 봅니다. 이것은 트리거링 SQL 조작에 의해 위반될 수도 있는 무결성 제한조건이 점검된 후에 수행됨에 주의하십시오. 결국 이것은 주로 응용프로그램도 수행할 수 있는 조작을 수행할 때 사용할 수 있습니다. 예를 들면 다음과 같습니다.

- 데이터베이스에서 후속 갱신 조작을 수행하십시오.
- 경보 지원 등의 조치는 데이터베이스 외부에서 수행하십시오. 데이터베이스 외부에서 수행된 조치는 트리거가 구간 복원되어도, 구간 복원되지 않습니다.

BEFORE와 AFTER 트리거의 서로 다른 특성으로 인해, BEFORE와 AFTER 트리거의 트리거 조치를 정의하는 데에는 서로 다른 SQL 조작 세트가 사용될 수 있습니다. 예를 들어, 갱신 조작은 BEFORE 트리거에서 허용되지 않는데, 이는 트리거 조치로 인해 무결성 제한조건이 위반되지 않는다는 보장이 없기 때문입니다. 사용자가 BEFORE와 AFTER에 지정할 수 있는 SQL 조작 세트는 548 페이지의 『트리거 조치』에 설명됩니다. 마찬가지로, 서로 다른 트리거 수준이 BEFORE와 AFTER 트리거에서 지원됩니다. 예를 들어, FOR EACH STATEMENT는 BEFORE 트리거에서 허용되지 않는데, 이는 트리거 조치에 의해 제한조건이 위반되지 않는다는 보장이 없기 때문이며, 즉 조작의 실패가 발생할 수 있기 때문입니다.

전이 변수

FOR EACH ROW 트리거를 수행할 때, 이 트리거가 현재 실행중인 영향을 받는 행 집합의 행의 컬럼 값을 참조할 필요가 있을 것입니다. 주제 테이블을 포함하여 데이터베이스의 테이블 컬럼을 참조하기 위해, 일반적인 SELECT문을 사용할 수 있습니다. FOR EACH ROW 트리거는 CREATE TRIGGER문의 REFERNCING 절에 지정할 수 있는 두 개의 전이 변수를 사용하여 현재 실행되고 있는 행의 컬럼을 참조할 수 있습니다. 두 가지 종류의 전이 변수가 있는데, *OLD*와 *NEW*로 지정되며, 두 가지 함께 상관 이름을 만들 수 있습니다. 여기에는 다음과 같은 의미가 있습니다.

OLD 상관 이름

행의 원래 상태, 즉 트리거 조치가 데이터베이스에 적용되기 전의 상태를 캡쳐하는 상관 이름을 지정합니다.

NEW 상관 이름

트리거 조치가 데이터베이스에 적용될 때 데이터베이스 행을 갱신하는 데 사용된 또는 사용되었던 값을 캡쳐하는 상관 이름을 지정합니다.

다음 예를 살펴 보십시오.

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N_ROW
  FOR EACH ROW MODE DB2SQL
  WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED
  AND N_ROW.ORDER_PENDING = 'N')
  BEGIN ATOMIC
    VALUES(ISSUE_SHIP_REQUEST(N_ROW.MAX_STOCKED -
                              N_ROW.ON_HAND,
                              N_ROW.PARTNO));
    UPDATE PARTS SET PARTS.ORDER_PENDING = 'Y'
    WHERE PARTS.PARTNO = N_ROW.PARTNO;
  END
```

위에서 주어진 OLD와 NEW 전이 변수의 정의에 기초하면, 모든 전이 변수가 모든 트리거에 정의될 수 있지는 않음이 분명합니다. 전이 변수는 트리거 이벤트의 종류에 따라 정의될 수 있습니다.

UPDATE

UPDATE 트리거는 OLD와 NEW 전이 변수를 모두 참조할 수 있습니다.

INSERT

INSERT 트리거는 NEW 전이 변수만을 참조할 수 있는데, 이것은 INSERT 조작의 활성화 전에 영향을 받은 행이 데이터베이스에 존재하지 않기 때문입니다. 즉, 트리거 조치가 데이터베이스에 적용되기 전의 오래된 값을 정의할 원래의 행 상태가 없습니다.

DELETE

DELETE 트리거는 OLD 전이 변수만을 참조할 수 있는데, 이는 삭제 조작에 지정된 새로운 값이 없기 때문입니다.

주: 전이 변수는 FOR EACH ROW 트리거에 대해서만 지정할 수 있습니다. FOR EACH STATEMENT 트리거에서, 전이 변수에 대한 참조는, 전이 변수가 참조하고 있는 영향을 받은 행 집합의 몇몇 행들을 지정하기에 충분하지 않습니다.

전이 테이블

FOR EACH ROW 및 FOR EACH STATEMENT 트리거 모두에서 영향을 받은 행 집합 전체를 참조해야 할 수도 있습니다. 예를 들어 트리거 내용에서 영향을 받은 행 집합에 총계를 적용해야 하는 경우(예를 들면 일부 컬럼 값의 MAX, MIN 또는 AVG)에 필요합니다. 트리거는 CREATE TRIGGER문의 REFERENCING절에 지정될 수 있는 두 개의 전이 변수를 사용하여 영향을 받은 행 집합을 참조할 수 있습니다. 전이변수와 마찬가지로 두 종류의 전이 테이블이 있는데, 이것은 OLD_TABLE과 NEW_TABLE로 지정되며 두 가지를 합하여 *table-name*이 되는데, 그 의미는 다음과 같습니다.

OLD_TABLE 테이블 이름
: 영향을 받은 행 집합의 원래 상태 즉, 트리거링 SQL 조작이 데이터베이스에 적용되기 전의 상태를 캡쳐하는 테이블 이름을 지정합니다.

NEW_TABLE 테이블 이름
: 트리거 조치가 데이터베이스에 적용될 때 데이터베이스의 행을 갱신하는 데 사용되는 값을 캡쳐하는 테이블 이름을 지정합니다.

예를 들면 다음과 같습니다.

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW_TABLE AS N_TABLE
  NEW AS N_ROW
  FOR EACH ROW MODE DB2SQL
  WHEN ((SELECT AVG (ON_HAND) FROM N_TABLE) > 35)
  BEGIN ATOMIC
    VALUES(INFORM_SUPERVISOR(N_ROW.PARTNO,
                             N_ROW.MAX_STOCKED,
                             N_ROW.ON_HAND));
  END
```

NEW_TABLE은 FOR EACH ROW 트리거의 경우에도 완전한 집합의 갱신된 행을 항상 가지고 있음에 주의하십시오. 트리거가 정의된 테이블에 트리거가 수행될 때, NEW_TABLE에는 해당 트리거를 활성화시킨 명령문으로부터 변경된 행이 들어갑니다. 그러나 NEW_TABLE에는 트리거 내부의 명령문에 의해 발생한 변경된 행은 들어 있지 않은데, 그 이유는 이것이 트리거의 별도 활성화를 유발하기 때문입니다.

전이 테이블은 읽기 전용입니다. 트리거 이벤트에 정의될 수 있었던 전이 변수의 종류를 정의하는 동일한 규칙이 전이 테이블에도 적용될 수 있습니다.

UPDATE

UPDATE 트리거는 OLD_TABLE과 NEW_TABLE 전이 테이블을 모두 참조할 수 있습니다.

INSERT

INSERT 트리거는 NEW_TABLE 전이 테이블만을 참조할 수 있는데, 이는 INSERT 조작의 활성화 전에 영향을 받은 행이 데이터베이스에 존재하지 않기 때문입니다. 즉 트리거 조치가 데이터베이스에 적용되기 전의 오래된 값을 정의하는 *행의 원래 상태*가 없습니다.

DELETE

DELETE 트리거는 삭제 조작에 지정된 새로운 값이 없기 때문에, OLD 전이 테이블만을 참조할 수 있습니다.

주: 전이 테이블을 AFTER 트리거의 두 가지 수준인 FOR EACH ROW와 FOR EACH STATEMENT 모두에 지정할 수 있다는 것은 중요합니다.

OLD_TABLE과 NEW_TABLE *table-name*의 범위는 트리거 내용입니다. 이 범위에서 이 이름은 스키마에 있을 수도 있는 동일한 완전하지 않은 *table-name*을 갖는 다른 테이블 이름보다 앞섭니다. 그러므로 OLD_TABLE이나 NEW_TABLE *table-name*이 예를 들어 X인 경우, SELECT문의 FROM절의 X(즉, 규정되지 않은 X)에 대한 참조는 트리거 작성자의 스키마의 X라는 이름의 테이블이 있어도 항상 전이 테이블을 참조합니다. 이 경우, 사용자는 스키마의 테이블 X를 참조하기 위해 완전한 이름을 사용해야 합니다.

트리거 조치

트리거 활성화는 관련 트리거 조치 수행을 발생시킵니다. 모든 트리거는 두 가지 구성요소를 갖는 정확히 하나의 트리거 조치로 구성됩니다.

- 옵션 *트리거 조치 조건* 또는 WHEN절
- *트리거 SQL문* 집합.

트리거 조치 조건은 트리거 조치가 실행하고 있는 명령문이나 행에 대하여 트리거 명령문 집합이 수행되는지 여부를 정의합니다. 트리거 명령문 집합은 발생한 이벤트의 결과 데이터베이스에서 트리거에 의해 수행되는 조치 집합을 정의합니다.

예를 들어, 다음의 트리거 조치는 트리거 SQL문 집합이 on_hand 컬럼이 max_stocked 컬럼 값의 10 퍼센트보다 작은 행에 대해서만 활성화되도록 지정합니다. 이 경우, 트리거 SQL문 집합은 issue_ship_request 함수 호출입니다.

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N_ROW
  FOR EACH ROW MODE DB2SQL
        WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED)
  BEGIN ATOMIC
    VALUES(ISSUE_SHIP_REQUEST(N_ROW.MAX_STOCKED -
                              N_ROW.ON_HAND,
                              N_ROW.PARTNO));
  END
```

트리거 조치 조건

『트리거 조치』에서 설명된대로, *트리거 조치 조건*은 트리거 조치 내에서 SQL문을 수행하기 위해 *참(true)*으로 평가되어야 하는 검색 조건을 지정하는 트리거 조치의 옵션 절입니다. WHEN절이 생략되면, 트리거 조치 내 SQL문은 항상 실행됩니다.

트리거가 FOR EACH ROW 트리거인 경우, 트리거 조치 조건은 각 행에 대하여 한 번 평가되고, 트리거가 FOR EACH STATEMENT 트리거인 경우, 명령문에 대하여 한 번 평가됩니다.

이 절에서는 트리거 대신 활성화 되는 조치를 조정하기 위해 사용할 수 있는 제어를 제공합니다. WHEN절의 유용함을 보여주는 한 예는 수입 값이 특정 범위 안에 또는 바깥에 있는 경우에만 트리거 조치가 활성화되는 데이터 종속 규칙을 강제하는 것입니다.

트리거 SQL문

트리거 SQL문 세트는 트리거를 활성화시킴으로써 유발되는 *실제 조치*를 수행합니다. 앞에서 설명되었듯이, 모든 SQL 조작이 모든 트리거에서 의미가 있는 것은 아닙니다. 트리거 활성화 시간이 BEFORE인지 또는 AFTER인지에 따라, 서로 다른 종류의 조작이 트리거 SQL문으로 적절할 수 있습니다.

트리거 SQL문 목록 및 BEFORE 및 AFTER 트리거에 대한 추가 정보는 *SQL 참조서* 책을 참조하십시오.

대부분의 경우, 트리거 SQL문이 음수의 리턴 코드를 리턴한 경우, 모든 트리거와 참조 제한조건과 함께 트리거한 SQL문이 구간 복원되고, 오류(SQLCODE -723(SQLSTATE 09000))가 리턴됩니다. 장애가 발생한 트리거 SQL문으로부터 트리거 이름, SQLCODE, SQLSTATE 및 많은 토큰들이 리턴됩니다. 트리거가 수행되고 있을 때 발생하는, 중요하거나 전 작업 단위를 구간 복원하는 오류 조건은 SQLCODE -723(SQLSTATE 09000)을 사용하여 리턴되지 않습니다.

SQL 트리거 명령문 내의 함수

사용자 정의 함수(UDF)를 포함한 함수들은 트리거 SQL문에서 호출될 수 있습니다. 다음 예를 살펴 보십시오.

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N_ROW
  FOR EACH ROW MODE DB2SQL
  WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED)
  BEGIN ATOMIC
    VALUES (ISSUE_SHIP_REQUEST (N_ROW.MAX_STOCKED - N_ROW.ON_HAND,
                                N_ROW.PARTNO));
  END
```

트리거 SQL문에 완전하지 않은 함수 이름을 사용한 함수 호출이 들어 있을 때, 이 함수 호출은 트리거 작성시 함수 경로에 기초하여 해석됩니다. 함수 해석에 대한 자세한 내용은 *SQL 참조서* 책을 참조하십시오.

UDF는 C나 C++ 프로그래밍 언어로 작성됩니다. 이들 언어로 작성하는 경우 논리 흐름의 제어, 오류 처리 및 복구, 시스템과 라이브러리 함수로의 액세스가 가능합니다. UDF에 대한 설명은 433 페이지의 『제15장 사용자 정의 함수(UDF)』의 내용을 참조하십시오. 이 기능을 통해 트리거 조치는 트리거가 활성화될 때 비 SQL 유형의 조작을 수행할 수 있습니다. 예를 들어, 그러한 UDF는 전자 우편 메시지를 송신하고, 이것으로 경보 메카니즘으로 작용합니다. 메시지와 같은 외부 조치는 확약 제어 하에 있지 않으며, 나머지 트리거 조치의 성패에 관계없이 수행됩니다.

또한 함수가 오류 발생을 나타내는 SQLSTATE를 리턴할 수도 있으며, 이것으로 트리거링 SQL문이 실패할 수도 있습니다. 이것은 사용자 정의 제한조건을 구현하는 한 가지 방법입니다. SIGNAL SQLSTATE문을 사용하는 것이 또 다른 방법입니다. 복잡한 사용자 정의 제한조건을 점검하는 수단으로 트리거를 사용하려면, 트리거 SQL문에서 `RAISE_ERROR` 내장 함수를 사용할 수 있습니다. 이 함수는 응용프로그램으로 사용자 정의 SQLSTATE(SQLCODE -438)를 리턴하는 데 사용할 수 있습니다. 이 함수의 호출과 사용에 대한 자세한 내용은 *SQL 참조서* 책을 참조하십시오.

예를 들어, HIREDATE가 직원이 근무를 시작한 날짜인, EMPLOYEE 테이블의 HIREDATE 컬럼에 관련된 어떤 규칙에 대하여 고려합시다.

- HIREDATE는 삽입한 날짜이거나 미래의 날짜여야 합니다.
- HIREDATE는 삽입한 날짜로부터 1년이 넘어서는 안됩니다.
- HIREDATE가 삽입한 날짜로부터 6개월에서 12개월 사이라면, `send_note`라는 UDF를 사용하여 인사 담당자에게 알리십시오.

다음의 트리거는 INSERT에서 이러한 규칙들을 모두 처리합니다.

```
CREATE TRIGGER CHECK_HIREDATE
  NO CASCADE BEFORE INSERT ON EMPLOYEE
  REFERENCING NEW AS NEW_EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
  VALUES CASE
    WHEN NEW_EMP.HIREDATE < CURRENT DATE
```

```
      THEN RAISE_ERROR('85001', 'HIREDATE has passed')
    WHEN NEW_EMP.HIREDATE - CURRENT DATE > 10000.
      THEN RAISE_ERROR('85002', 'HIREDATE too far out')
    WHEN NEW_EMP.HIREDATE - CURRENT DATE > 600.
      THEN SEND_MOTE('persmgr',NEW_EMP.EMPNO,'late.txt')
    END;
  END
```

트리거 연쇄

트리거 SQL문을 수행하면, 이것은 다른 또는 심지어 동일한 트리거의 이벤트가 발생하도록 하여, 다시 이것이 다른(또는 같은 트리거의 두 번째 인스턴스) 트리거를 활성화시킬 수 있습니다. 그러므로, 하나의 트리거를 활성화 시킴으로써 하나 이상의 다른 트리거를 연쇄적으로 활성화시킬 수 있습니다.

지원되는 트리거 연쇄의 런타임 깊이 레벨은 16입니다. 트리거 레벨 17이 활성화되면 SQLCODE -724(SQLSTATE 54038)가 리턴되고, 트리거한 명령문이 구간 복원됩니다.

참조 제한조건과의 상호작용

위에서 설명한대로, 트리거 이벤트는 참조 제한조건 강제 실행으로 인한 변경의 결과일 수 있습니다. 예를 들어, 두 개의 테이블 DEPT와 EMP가 있고, DEPT를 삭제하거나 갱신함으로써 참조 무결성 제한조건에 의해 EMP에 대한 삭제나 갱신이 발생하면, EMP에 정의된 삭제나 갱신 트리거는 DEPT에 정의된 참조 제한조건의 결과로 활성화됩니다. EMP에 대한 트리거는 활성화 시간에 따라, EMP의 행에 대한 삭제(ON DELETE CASCADE의 경우)나 갱신(ON DELETE SET NULL의 경우)의 전이나 후에 실행됩니다.

복수 트리거 정렬

CREATE TRIGGER문을 사용하여 트리거가 정의되면, 이들의 작성 시간이 시간소인의 형태로 데이터베이스에 등록됩니다. 이 시간소인 값은 동시에 수행되어야 하는 트리거가 둘 이상 있을 때 트리거 활성화의 순서를 정할 때 사용됩니다. 예를 들어, 시간소인은 같은 이벤트와 같은 활성화 시간을 갖는 동일한 주제 테이블에 대하여 둘 이상의 트리거가 있을 때 사용됩니다. 시간소인은 또한 트리거 조치

에 의해 직접적으로 또는 간접적으로(즉, 다른 참조 제한조건에 의해 순환적으로) 발생한 트리거 이벤트와 참조 제한조건에 의해 활성화된 AFTER 트리거가 둘 이상 있을 때 사용됩니다. 다음의 두 가지 트리거를 살펴 보십시오.

```
CREATE TRIGGER NEW_HIRED
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS
    SET NBEMP = NBEMP + 1;
  END;

CREATE TRIGGER NEW_HIRED_DEPT
   AFTER INSERT ON EMPLOYEE
   REFERENCING NEW AS EMP
   FOR EACH ROW MODE DB2SQL
     BEGIN ATOMIC
       UPDATE DEPTS
       SET NBEMP = NBEMP + 1
       WHERE DEPT_ID = EMP.DEPT_ID;
     END;
```

위의 트리거들은 사용자가 직원 테이블에 INSERT 조작을 수행할 때 활성화됩니다. 이 경우, 이들의 작성 시간소인은 위의 두 트리거 중 어떤 것이 먼저 활성화될 것인지를 정의합니다.

트리거 활성화는 시간소인 값의 오름차순으로 수행됩니다. 이렇게 데이터베이스에 새로 추가된 트리거는 이전에 정의된 다른 트리거들이 모두 수행된 후에 수행됩니다.

오래된 트래거는 새로운 트리거 전에 활성화되어, 새로운 트리거가 데이터베이스에 영향을 미치는 변경에 대하여 증분 추가로 사용될 수 있습니다. 예를 들어, 트리거 T1의 트리거 SQL문이 테이블 T에 새로운 행을 삽입한 경우, T1 다음에 수행되는 트리거 T2의 트리거 SQL문은 고유한 값을 사용하여 T의 같은 행을 갱신하는 데 사용될 수 있습니다. 작성 시간의 오름차순으로 트리거를 활성화시킴으로써, 새로운 트리고 조치가 그보다 앞선 트리거 활성화의 결과를 반영하는 데이터베이스 수행되도록 할 수 있습니다.

트리거, 제한조건, UDT, UDF 및 LOB간의 시너지

다음 절에서는 그러한 응용프로그램 구조의 모델로 트리거와 제한조건을 활용하는 방법에 대하여 설명합니다. 트리거를 사용하여 다음을 수행할 수 있습니다.

- 이 구조들의 정보를 추출하여, 이들 구조가 테이블 컬럼에 명시적으로 보관될 수 있도록 합니다(구조 내에 숨어 있는 대신).
- 응용프로그램 도메인에서 이 구조를 제어하는 무결성 규칙을 정의합니다.
- 특별한 구조 값 하에서 취할 수 있는 중요한 조치를 표현합니다.

정보 추출

LOB 값에서, 완전한 전자 우편이 ELECTRONIC_MAIL 테이블의 컬럼 MESSAGE에 저장됩니다. 전자 우편을 조작하기 위해서, UDF가 SQL문 안에서 그러한 정보를 필요로할 때마다 메시지 컬럼으로부터 정보를 추출합니다.

조회는 정보를 추출하지 않고, 이를 명시적으로 테이블 컬럼으로 저장함에 주의하십시오. 이것이 완료되면, UDF가 반복적으로 호출되지 않아서 뿐만 아니라 이제 추출한 정보에 색인을 정의할 수 있으므로, 조회 성능이 향상됩니다.

트리거를 사용하여, 사용자는 새로운 전자 우편이 데이터베이스에 저장될 때마다 이 정보를 추출할 수 있습니다. 이것을 이루기 위해 다음과 같이 새로운 컬럼을 ELECTRONIC_MAIL 테이블에 추가하고, BEFORE 트리거를 정의하여 다음과 같이 해당 정보를 추출합니다.

```
ALTER TABLE ELECTRONIC_MAIL
  ADD COLUMN SENDER    VARCHAR (200)
  ADD COLUMN RECEIVER  VARCHAR (200)
  ADD COLUMN SENT_ON   DATE
  ADD COLUMN SUBJECT   VARCHAR (200)

CREATE TRIGGER EXTRACT_INFO
  NO CASCADE BEFORE INSERT ON ELECTRONIC_MAIL
  REFERENCING NEW AS N
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    SET N.SENDER = SENDER(N.MESSAGE);
    SET N.RECEIVER = RECEIVER(N.MESSAGE);
    SET N.SENT_ON = SENDING_DATE(N.MESSAGE);
    SET N.SUBJECT = SUBJECT(N.MESSAGE);
  END
```

이제 새로운 전자 우편이 메시지 컬럼에 삽입될 때마다, 송신자, 수신자, 송신된 날짜 및 주제가 메시지에서 추출되어 별도의 컬럼에 저장됩니다.

테이블에 대한 조작 금지

전자우편 주소가 틀렸을 경우, 송신한 우편을 송신되지 못하도록, 배달되지 않고 다시 사용자에게 리턴되어, 전자 우편 테이블에 저장되지 않도록 하고자 한다고 가정하십시오.

이를 위해서 사용자는 특정 SQL INSERT문의 실행을 금지해야 합니다. 이것을 수행하기 위한 방법은 두 가지가 있습니다.

- 전자우편의 주제가 *배달되지 않은 우편*일 때마다 오류를 나타내는 BEFORE 트리거를 정의하십시오.

```
CREATE TRIGGER BLOCK_INSERT
  NO CASCADE BEFORE INSERT ON ELECTRONIC_MAIL
  REFERENCING NEW AS N
  FOR EACH ROW MODE DB2SQL
  WHEN (SUBJECT(N.MESSAGE) = 'undelivered mail')
  BEGIN ATOMIC
    SIGNAL SQLSTATE '85101' ('Attempt to insert undelivered mail');
  END
```

- 새로운 컬럼 주제 값이 *미배달 우편*과는 다르도록 점검 제한조건을 정의하십시오.

```
ALTER TABLE ELECTRONIC_MAIL
  ADD CONSTRAINT NO_UNDELIVERED
  CHECK (SUBJECT <> 'undelivered mail')
```

제한조건의 선언적 성격의 이점으로 인해 제한조건은 일반적으로 트리거 대신 정의되어야 합니다.

비지니스 규칙 정의

사용자의 회사에서는 고객의 불만을 다루는 모든 전자 우편의 CC 목록에 마케팅 관리자 Mr. Nelson이 있어야 한다고 가정하십시오. 이것은 규칙이기 때문에, 이것을 다음 중 하나와 같은 제한조건으로 표현하고자 할 수 있습니다(이것을 확인할 CC_LIST UDF의 존재를 가정)

```
ALTER TABLE ELECTRONIC_MAIL ADD
  CHECK (SUBJECT <> 'Customer complaint' OR
         CONTAINS (CC_LIST(MESSAGE), 'nelson@vnet.ibm.com') = 1)
```

그러나 그러한 제한조건은 cc 목록에 마케팅 관리자가 없는 고객 불만을 다룬 전자 우편의 삽입을 금하게 됩니다. 이것은 분명 회사의 비지니스 규칙이 의도하는 바는 아닙니다. 원하는 것은 마케팅 관리자로 복사되지 않았던 고객 불만을 다룬 전자 우편을 마케팅 관리자에게 이송하는 것입니다. 그러한 비지니스 규칙은, 선언적 제한조건으로는 표현할 수 없는 조치를 필요로 하기 때문에, 트리거로만 표현될 수 있습니다. 트리거는 E_MAIL 유형의 매개변수와 문자열을 갖는 SEND_NOTE 함수의 존재를 가정합니다.

```
CREATE TRIGGER INFORM_MANAGER
  AFTER INSERT ON ELECTRONIC_MAIL
  REFERENCING NEW AS N
  FOR EACH ROW MODE DB2SQL
  WHEN (N.SUBJECT = 'Customer complaint' AND
    CONTAINS (CC_LIST(MESSAGE), 'nelson@vnet.ibm.com') = 0)
  BEGIN ATOMIC
    VALUES(SEND_NOTE(N.MESSAGE, 'nelson@vnet.ibm.com'));
  END
```

조치 정의

이제 사용자의 일반 관리자는 최근 72시간 동안 세장 이상의 불만을 송신한 고객의 이름을 별도의 테이블에 보관하고자 합니다. 일반 관리자는 또한 고객의 이름이 이 테이블에 한 번 이상 삽입될 때마다 이에 대한 통지를 받고자 합니다.

그러한 조치를 정의하려면, 다음을 정의하십시오.

- UNHAPPY_CUSTOMERS 테이블:

```
CREATE TABLE UNHAPPY_CUSTOMERS (
   NAME              VARCHAR (30),
  EMAIL_ADDRESS  VARCHAR (200),
  INSERTION_DATE DATE)
```

- 최근 3일 동안에 셋 이상의 메시지가 수신된 경우, UNHAPPY_CUSTOMERS에 자동으로 행이 삽입되도록 하는 트리거(NAME 컬럼과 E_MAIL_ADDRESS 컬럼을 포함하는 CUSTOMERS 테이블의 존재를 가정):

```
CREATE TRIGGER STORE_UNHAPPY_CUST
  AFTER INSERT ON ELECTRONIC_MAIL
  REFERENCING NEW AS N
  FOR EACH ROW MODE DB2SQL
  WHEN (3 <= (SELECT COUNT(*)
  FROM ELECTRONIC_MAIL
            WHERE SENDER = N.SENDER
```

```
                 AND SENDING_DATE(MESSAGE) > CURRENT DATE - 3 DAYS)
        )
    BEGIN ATOMIC
      INSERT INTO UNHAPPY_CUSTOMERS
      VALUES ((SELECT NAME
      FROM CUSTOMERS
      WHERE E_MAIL_ADDRESS = N.SENDER), N.SENDER, CURRENT DATE);
    END
```

• 같은 고객이 UNHAPPY_CUSTOMERS에 두 번 이상 삽입되는 경우, 일반 관리자에게 노트를 송신하는 트리거(입력으로 2자 문자열을 받아들이는 SEND_NOTE 함수의 존재를 가정):

```
  CREATE TRIGGER INFORM_GEN_MGR
    AFTER INSERT ON UNHAPPY_CUSTOMERS
    REFERENCING NEW AS N
    FOR EACH ROW MODE DB2SQL
    WHEN (1 <(SELECT COUNT(*)
              FROM UNHAPPY_CUSTOMERS
              WHERE EMAIL_ADDRESS = N.EMAIL_ADDRESS)
        )
    BEGIN ATOMIC
      VALUES(SEND_NOTE('Check customer:' CONCAT N.NAME,
                       'bigboss@vnet.ibm.com'));
    END
```

제5부 DB2 프로그래밍 고려사항

© Copyright IBM Corp. 1993, 2000

제17장 복잡한 환경에서의 프로그래밍

자국어 지원 고려사항 560
조합 순서 개요 560
조합 순서 560
조합 순서 정렬 순서: EBCDIC 및 ASCII 예 564
조합 순서 지정 565
코드 페이지 값 구하기 566
응용프로그램에서 로케일 구하기 567
DB2가 로케일을 구하는 방법. 568
자국어 지원 응용프로그램 개발 568
SQL문 코딩 569
원격 저장 프로시듀어와 UDF 코딩 571
혼합 코드 페이지 환경에서 패키지 이름 고려사항. 571
사전 처리 컴파일 및 바인딩 572
응용프로그램 실행 572
주의사항. 572
다른 코드 페이지 간의 변환 573
DBCS 문자 집합. 577
확장 UNIX 코드(EUC) 문자 세트 . . . 578
DBCS 환경에서 CLI/ODBC/JDBC/SQLJ 프로그램 수행 579
일본어 및 대만어 EUC 및 UCS-2 코드 세트 변환 581
혼합 EUC 및 2 바이트 클라이언트 및 데이터베이스 고려사항 583
대만어 사용자들을 위한 고려사항 . . 584
일본어 또는 대만어 EUC 응용프로그램 개발 584
혼합 코드 세트 환경에서의 개발 . . . 586
유니코드(UCS-2) 데이터베이스에 연결된 응용프로그램 596
다중 사이트 갱신시 고려사항 597
원격 작업 단위(RUOW) 597
다중 사이트 갱신. 598
다중 사이트 갱신 사용 시점 599
다중 사이트 갱신 응용프로그램을 위한 SQL 코딩 599
다중 사이트 갱신 응용프로그램 사전 처리 컴파일. 602
다중 사이트 갱신 응용프로그램의 구성 매개변수 지정 604
다중사이트 갱신 제한사항 605
Host 또는 AS/400 서버에 액세스 606
다중 스레드 데이터베이스 액세스 607
다중 스레드 사용시 권장사항 608
코드 페이지 및 국가 코드에 대하여 작업하는 다중 스레드 UNIX 응용프로그램. . 609
다중 스레드 사용시 잠재적 함정. . . . 610
복수 문맥에서 교착 상태 방지 . . . 611
동시 트랜잭션 612
동시 트랜잭션 사용시 잠재적 함정 . . . 613
동시 트랜잭션의 교착 상태 방지. . . 614
X/Open XA 인터페이스 프로그래밍 고려사항 615
응용프로그램 링크 619
네트워크에 걸쳐 있는 대용량의 데이터에 대한 작업 619

© Copyright IBM Corp. 1993, 2000

자국어 지원 고려사항

이 절에서는 사용자의 응용프로그램에서 고려해야 하는 자국어 지원(NLS)에 관한 주제를 설명합니다. 논의되는 주요 주제는 다음과 같습니다.

- 조합 순서
- 다른 코드 페이지 간의 변환
- 코드 페이지 값 구하기
- 응용프로그램에서 로케일 구하기
- 자국어 지원 응용프로그램 개발

조합 순서 개요

조합 순서

데이터베이스 관리 프로그램은 *조합 순서*를 사용하여 문자 데이터를 비교합니다. 이것은 특정 문자가 다른 것보다 상위인지, 하위인지 또는 같은지 여부를 판별하는 문자 세트에 대한 정렬입니다.

주: FOR BIT DATA 속성이나 BLOB 데이터를 사용하여 정의된 문자열 데이터는 2진 정렬 순서를 사용하여 정렬됩니다.

예를 들어 특정 문자의 소문자와 대문자 버전이 동등하게 정렬되었음을 나타내는 데 조합 순서가 사용될 수 있습니다.

데이터베이스 관리 프로그램을 통해 데이터베이스는 사용자 정의 조합 순서를 사용하여 작성될 수 있습니다. 다음 절에서는 데이터베이스에 대한 특정 조합 순서를 판별하고 구현하는 데 도움이 되는 정보를 제공합니다.

데이터베이스에 있는 각각의 단일 바이트 문자는 내부적으로 0과 255(16진수 표기법으로는 X'00'과 X'FF') 사이에 있는 고유 숫자로 표현됩니다. 이 숫자는 문자의 *코드 포인트*라고 하며, 세트에서 문자에 숫자를 할당한 것을 집합적으로 *코드 페이지*라고 합니다. 조합 순서는 정렬된 각 문자의 원하는 위치와 코드 포인트간의 맵핑입니다. 위치의 숫자 값을 조합 순서에서 문자의 *가중치*라고 합니다. 가장 간단한 조합 순서에서 가중치는 코드 포인트와 동일합니다. 이것을 *동일 순서*라고 합니다.

예를 들어, B(X'42') 및 b(X'62') 문자를 고려해 보십시오. X'42'(B)의 정렬 가중치를 모두 사용하면(조합 순서 테이블에 따라) 동일하게 조합합니다. B의 정렬 가중치는 X'9E'이고, b의 정렬 가중치는 X'9D'이며, b는 B 전에 정렬됩니다. 실제 가중치는 사용된 조합 순서 테이블에 따라 달라지며 이 테이블은 코드 세트 및 로케일에 따라 달라집니다. 조합 순서 테이블은 코드 포인트를 정의하는 코드 페이지 테이블과 같지 않습니다.

다음 예를 살펴 보십시오. A에서 Z까지의 ASCII 문자는 X'41'에서 X'5A'로 표현됩니다. 이런 문자가 연속적으로 정렬되는 조합 순서(중단 문자 없이)를 설명하려면 X'41', X'42', ... X'59', X'5A'를 기록하면 됩니다.

복수 바이트 문자의 16진수 값 또한 가중치로 사용됩니다. 예를 들어 X'8260', X'8261'은 2 바이트 A와 B 문자의 코드 포인트입니다. 이 경우 X'82', X'60' 및 X'61'의 조합 가중치가 이들 두 문자의 코드 포인트에 따라 이들 문자를 정렬하는데 사용됩니다.

조합 순서의 가중치 값은 고유할 필요는 없습니다. 예를 들어 대문자와 소문자에 같은 가중치를 부여할 수도 있습니다.

조합 순서 지정은 조합 순서에서 256 포인트 전부의 가중치를 제공할 때 간단해질 수 있습니다. 각 문자의 가중치는 문자의 코드 포인트를 사용하여 판별할 수 있습니다. *n*번째 바이트(0에서 시작)에 코드 포인트 *n*의 가중치가 들어 있는 256 바이트 문자열의 데이터베이스 관리 프로그램 조합 순서를 지정할 때 사용되는 방법입니다.

모든 경우 DB2는 데이터베이스 작성 시간에 지정된 조합 테이블을 사용합니다. 코드 포인트 테이블에 나타난 대로 복수 바이트 문자를 정렬하려면 데이터베이스를 작성할 때 IDENTITY를 조합 순서로 지정해야 합니다.

주: GRAPHIC 필드의 DBCS 문자에 대해 정렬 순서는 항상 IDENTITY입니다.

문자 비교: 일단 조합 순서가 정해지면, 코드 포인트 값을 직접 비교하는 대신 두 문자의 가중치를 비교하여 문자 비교가 이루어집니다.

고유하지 않은 가중치가 사용된 경우, 동일하지 않은 문자들이 동일하게 비교될 수 있습니다. 이 때문에 문자열 비교는 두 단계로 이루어져야 합니다.

1. 가중치를 기초로 각 문자열의 문자를 비교합니다.
2. 단계 1의 결과가 동일하다면 코드 포인트 값에 기초하여 각 문자열의 문자를 비교합니다.

조합 순서가 256개의 고유한 가중치를 가지고 있다면, 첫번째 단계만을 수행합니다. 조합 순서가 동일한 순서라면, 두 번째 단계만 수행합니다. 어떤 경우든 성능이 저하되지 않습니다.

문자 비교에 대해서는 *SQL 참조서*를 참조하십시오.

대소문자에 독립적인 비교: 대소문자에 관계없이 문자를 비교하려면 TRANSLATE 함수를 사용하여 대소문자 혼합 컬럼 데이터를 대문자로 변환함으로써 이를 선택 및 비교할 수 있습니다(비교용으로만 사용). 다음의 데이터를 살펴보십시오.

```
Abel
abels
ABEL
abel
ab
Ab
```

다음 SELECT문은

```
SELECT c1 FROM T1 WHERE TRANSLATE(c1) LIKE 'AB%'
```

다음을 리턴합니다.

```
ab
Ab
abel
Abel
ABEL
abels
```

"v1"을 작성할 때 다음 SELECT문을 지정하여 대문자로 뷰에 대해 전부 비교하고 대소문자로 INSERT 테이블을 요청할 수 있습니다.

```
CREATE VIEW v1 AS SELECT TRANSLATE(c1) FROM T1
```

데이터베이스 레벨에서 조합 순서를 `sqlecrea` - Create Database API의 일부로 설정할 수 있습니다. 이를 통해 사용자는 'A' 전에 'a'가 처리될 것인지, 'a' 전에

'A'가 처리될 것인지 또는 같은 가중치로 처리될 것인지를 결정할 수 있습니다. 이것은 ORDER BY절을 사용하여 조합하거나 정렬할 때에도 마찬가지입니다. "A"와 "a"는 모든 면에서 동등하므로 "A"는 항상 "a" 전에 옵니다. 유일한 정렬 기준은 16진수 값입니다.

그러므로

```
SELECT c1 FROM T1 WHERE c1 LIKE 'ab%'
```

다음을 리턴합니다.

```
ab
abel
abels
```

그리고

```
SELECT c1 FROM T1 WHERE c1 LIKE 'A%'
```

다음을 리턴합니다.

```
Abel
Ab
ABEL
```

다음 명령문은

```
SELECT c1 FROM T1 ORDER BY c1
```

다음을 리턴합니다.

```
ab
Ab
abel
Abel
ABEL
abels
```

그러므로 sqlecrea뿐만 아니라 스칼라 함수인 TRNSLATE()를 사용하는 것에 대해 고려해 볼 수 있습니다. sqlecrea를 통해서만 조합 순서를 지정할 수 있습니다. 명령행 프로세서(CLP)에서 조합 순서를 지정할 수 없습니다. TRANSLATE() 함수에 대한 자세한 정보는 *SQL 참조서*를 참조하십시오. sqlecrea에 대한 자세한 정보는 *Administrative API Reference*를 참조하십시오.

또한 다음과 같이 UCASE 함수를 사용할 수 있는데, DB2가 select의 색인을 사용하는 대신 테이블 스캔을 수행함에 주의하십시오.

```
SELECT * FROM EMP WHERE UCASE(JOB) = 'NURSE'
```

조합 순서 정렬 순서: EBCDIC 및 ASCII 예

데이터베이스에 있는 데이터가 정렬되는 순서는 데이터베이스에 대해 정의된 조합 순서에 따라 달라집니다. 예를 들어, 데이터베이스 A가 EBCDIC 코드 페이지의 기본 조합 순서를 사용하고 데이터베이스 B가 ASCII 코드 페이지의 기본 조합 순서를 사용한다고 하십시오. 이런 두 데이터베이스에서의 정렬 순서는 그림19에서와 같이 다릅니다.

```
SELECT.....
  ORDER BY COL2

EBCDIC-Based Sort         ASCII-Based Sort

COL2                      COL2
----                      ----
V1G                       7AB
Y2W                       V1G
7AB                       Y2W
```

그림 19. EBCDIC-기반 순서에서의 정렬 순서 및 ASCII-기반 순서에서의 정렬 순서가 서로 다른 경우의 예

이와 비슷하게 데이터베이스에서의 문자 비교는 그 데이터베이스에 대해 정의된 조합 순서에 따라 달라집니다. 그러므로 데이터베이스 A가 EBCDIC 코드 페이지의 기본 조합 순서를 사용하고 데이터베이스 B는 ASCII 코드 페이지의 기본 조합 순서를 사용하면 두 데이터베이스에서의 두 문자 비교 결과는 달라집니다. 565 페이지의 그림20은 그 차이를 보여줍니다.

```
SELECT.....
  WHERE COL2 > 'TT3'

EBCDIC-Based Results     ASCII-Based Results

COL2                     COL2
----                     ----
TW4                      TW4
X72                      X72
39G
```

그림 20. EBCDIC-기반 순서에서의 문자 비교 및 ASCII-기반 순서에서의 문자 비교가 서로 다른 경우의 예

연합 데이터베이스를 작성하는 경우 조합 순서가 데이터 소스에서 조합 순서와 일치하도록 지정하십시오. 이 접근 방법은『푸쉬다운』기회를 극대화하고 조회 성능을 향상시킬 수도 있습니다. 푸쉬다운 분석, 조합 순서 그리고 조회 성능에 대한 자세한 내용은 *관리 안내서 : 구현*을 참조하십시오.

조합 순서 지정

데이터베이스의 조합 순서는 데이터베이스 작성시에 지정됩니다. 일단 데이터베이스가 작성되면, 조합 순서는 변경될 수 없습니다.

CREATE DATABASE API는 데이터베이스 설명자 블록(SQLEDBDESC)이라고 하는 데이터 구조를 수용합니다. 이 구조 내에서 사용자 자신의 조합 순서를 정의할 수 있습니다.

데이터베이스의 조합 순서를 지정하려면 다음과 같이 하십시오.

- 원하는 SQLEDBDESC 구조를 전달하거나
- 널(NULL) 포인터를 전달하십시오. 현재의 국가 코드와 코드 페이지에 기초한 운영 체제의 조합 순서가 사용됩니다. 이것은 SQLDBCSS를 SQL_CS_SYSTEM (0)으로 지정한 것과 동일합니다.

SQLEDBDESC 구조에는 다음과 같은 것들이 포함됩니다.

SQLDBCSS

데이터베이스 조합 순서의 소스를 나타내는 4 바이트 정수. 유효한 값은 다음과 같습니다.

SQL_CS_SYSTEM

현재의 국가 코드와 코드 페이지에 기초한 운영 체제의 조합 순서가 사용됩니다.

SQL_CS_USER

조합 순서는 SQLDBUDC 필드의 값에 의해 지정됩니다.

SQL_CS_NONE

조합 순서는 동일 순서입니다. 문자열은 첫번째 바이트부터 시작하여 바이트 단위로 비교되고, 단순 코드 포인트 비교를 수행합니다.

주: 이 상수들은 SQLENV include 파일에 정의됩니다.

SQLDBUDC

256 바이트 필드. n번째 바이트에는 데이터베이스의 코드 페이지의 n번째 문자의 정렬 가중치가 들어 있습니다. SQLDBCSS는 SQL_CS_USER와 같지 않으며, 이 필드는 무시됩니다.

샘플 조합 순서: 기본적인 워크스테이션 조합 순서 대신 EBCDIC 조합 순서를 사용하는 데이터베이스 작성을 위해 여러개의 샘플 조합 순서가 include 파일로 제공됩니다.

이 include 파일들의 조합 순서는 SQLEDBDESC 구조의 SQLDBUDC 필드에 지정될 수 있습니다. 이들은 다른 조합 순서 설정을 위한 모델로도 사용될 수 있습니다.

조합 순서가 포함된 include 파일에 대한 정보는 다음 절을 참조하십시오.

- C/C++에 대해서는 663 페이지의『C 및 C++에 대한 파일』
- COBOL에 대해서는 762 페이지의『COBOL의 Include 파일』
- FORTRAN에 대해서는 789 페이지의『FORTRAN에 대한 포함 파일』.

코드 페이지 값 구하기

*응용프로그램 코드 페이지*는 데이터베이스가 연결될 때 사용중인 환경으로부터 구할 수 있습니다. DB2CODEPAGE 레지스트리 변수가 설정되면, 그 값은 응용프로그램 코드 페이지로 취해집니다. 그러나 DB2가 운영 체제로부터 적절한 코드 페

이지 값을 판별하기 때문에 DB2CODEPAGE 레지스트리 변수를 설정할 필요가 없습니다. DB2CODEPAGE 레지스트리 변수를 잘못된 값에 설정하면 예측할 수 없는 결과가 발생할 수도 있습니다.

*데이터베이스 코드 페이지*는 데이터베이스 작성시 명시적으로 또는 기본적으로 지정된 값으로부터 구해집니다. 예를 들어, 다음은 *사용중인 환경*을 여러 운영 환경에서 판별하는 방법을 정의합니다.

UNIX
UNIX 기반 운영 체제에서 사용중인 환경은 언어, 지역 그리고 코드 세트에 대한 정보가 들어 있는 로케일 설정에서 결정됩니다.

OS/2
OS/2에서 1차와 2차 코드 페이지는 `CONFIG.SYS` 파일에 지정됩니다. `chcp` 명령을 사용하여 해당 세션 내에서 코드 페이지를 표시하고 동적으로 변경할 수 있습니다.

Windows 32 비트 운영 체제
모든 Windows 32 비트 운영 체제의 경우, DB2CODEPAGE 환경 변수가 설정되지 않은 경우, 코드 페이지는 레지스트리의 ANSI 코드 페이지 설정으로부터 구합니다.

코드 페이지 값의 완전한 환경 맵핑 목록에 대해 *관리 안내서*를 참조하십시오.

응용프로그램에서 로케일 구하기

로케일은 Windows와 UNIX 기반 시스템에서 서로 다른 방식으로 구현됩니다. UNIX 기반 시스템에는 두 가지 로케일이 있습니다.

- 환경 로케일을 사용하여 사용자는 사용하려는 언어, 통화 기호 등을 지정할 수 있습니다.
- 프로그램 로케일에는 실행 중인 프로그램의 현재 언어, 통화 기호 등이 들어 있습니다.

Windows에서 문화적 기본 설정은 제어판의 지역적 설정을 통해 설정됩니다. 그러나, UNIX 기반 시스템에서와 같은 환경 로케일은 없습니다.

프로그램이 시작되면 기본값으로 C 로케일을 갖습니다. 이는 환경 로케일 사본을 갖지 *않습니다*. 프로그램 로케일을 ″C″가 아닌 다른 로케일로 설정하는 경우, DB2 Universal Database는 사용자의 현재 프로그램 로케일을 사용하여 응용프로그램 환경을 위한 코드 페이지와 지역 설정값을 결정합니다. 그렇지 않은 경우, 이 값들은 운영 체제 환경으로부터 구해집니다. `setlocale()`은 스레드에 안전하지 않으며, 사용자가 응용프로그램에서 `setlocale()`을 발행하는 경우 전체 프로세스에 대하여 새로운 로케일이 설정됩니다.

DB2가 로케일을 구하는 방법

UNIX 기반 시스템에서 DB2에 의해 사용되는 사용중인 로케일은 로케일의 LC_CTYPE 부분으로부터 판별됩니다. 자세한 내용은 사용자 운영 체제의 NLS 관련 문서를 참조하십시오.

- 프로그램 로케일의 LC_CTYPE에 ″C″ 값이 아닌 다른 값이 들어 있는 경우, DB2는 이 값을 사용하여 해당 코드 페이지에 응용프로그램 코드 페이지를 맵핑함으로써 응용프로그램 코드 페이지를 판별합니다.
- LC_CTYPE에 ″C″ 값(″C″ 로케일)이 들어 있는 경우 DB2는 `setlocale()` 함수를 사용하여 환경 로케일에 따라 프로그램 로케일을 설정합니다.
- LC_CTYPE에 여전히 ″C″의 값이 있으면 DB2는 미국 영어 환경과 코드 페이지 819(ISO 8859-1)를 기본값으로 간주합니다.
- LC_CTYPE에는 더 이상 ″C″의 값이 없으면, 새로운 값이 해당 코드 페이지를 맵핑하는 데 사용됩니다. 특정 플랫폼의 기본 로케일에 대한 내용은 *관리 안내서*를 참조하십시오. 특정 플랫폼에서 응용프로그램을 구축하는 것에 대한 자세한 내용은 *응용프로그램 빌드 안내서*을 참조하십시오.

자국어 지원 응용프로그램 개발

정적 SQL문의 상수 문자열은 바인드될 때 응용프로그램 코드 페이지로부터 데이터베이스 코드 페이지로 변환되며, 이 데이터베이스 코드 페이지 표기법으로 실행 시 사용됩니다. 이러한 변환이 바람직하지 않아 이를 방지하려는 경우, 문자열 상수 대신 호스트 변수를 사용할 수 있습니다.

프로그램에 상수 문자열이 있으면 동일한 코드 페이지를 사용하여 응용프로그램을 사전 처리 컴파일, 바인드, 컴파일 그리고 실행하도록 강력히 권합니다. 유니코드 데이터베이스에 대해 문자열 상수 대신 호스트 변수를 사용해야 합니다. 이것은 서

버에 의한 데이터 변환이 바인드와 실행 단계 모두에서 발생할 수 있기 때문입니다. 이것은 상수 문자열이 프로그램 내에서 사용되었을 때 문제가 될 수 있습니다. 이렇게 삽입된 문자열은 바인드 단계동안 유효한 코드 페이지에 근거하여 바인드시 변환됩니다. 7 비트 ASCII 문자는 DB2 Universal Database에 의해 지원되는 모든 코드 페이지에 공통이며, 문제를 발생시키지 않습니다. 비 ASCII 문자의 경우, 사용자들은 반드시 동일한 사용중인 코드 페이지를 사용하여 바인딩 및 실행을 함으로써, 같은 변환 테이블을 사용하도록 하십시오. 응용프로그램이 사용중인 코드 페이지를 판별하는 방식에 대한 설명은 566 페이지의 『코드 페이지 값 구하기』의 내용을 참조하십시오.

응용프로그램에 의해 얻어지는 외부 데이터는 응용프로그램 코드 페이지에 있는 것으로 가정됩니다. 여기에는 파일이나 사용자 입력으로부터 얻어진 데이터도 포함됩니다. 응용프로그램 외부의 소스로부터 얻어진 데이터는 응용프로그램과 같은 코드 페이지를 사용하는지 확인하십시오.

사용자의 C나 C++ 응용프로그램에서 그래픽 데이터를 사용하는 호스트 변수를 사용하는 경우, 고려해야 할 특별한 사전 처리 컴파일러, 응용프로그램 성능 및 응용프로그램 설계시 주의점 등이 있습니다. 이런 고려사항에 대한 상세한 설명은 694 페이지의 『C 및 C++에서의 그래픽 호스트 변수 처리』를 참조하십시오. 사용자 응용프로그램에서 EUC 코드 세트를 다루는 경우, 사용자가 고려해야 하는 지침은 581 페이지의 『일본어 및 대만어 EUC 및 UCS-2 코드 세트 변환』의 내용을 참조하십시오.

SQL문 코딩

SQL문 코딩은 언어와는 개별적입니다. SQL 키워드는 대문자, 소문자 또는 대소문자가 혼합되어 입력될 수 있지만 이 책에서 설명된대로 입력되어야 합니다. SQL문에서 발생하는 데이터베이스 오브젝트, 호스트 변수 및 프로그램 레이블의 이름에는 사용자의 코드 페이지가 지원하는 확장 문자 세트 이외의 문자가 포함되어서는 안됩니다. 확장 문자 세트에 대한 자세한 내용은 *SQL 참조서*를 참조하십시오.

서버는 파일 이름을 변환하지 않습니다. 파일 이름을 코딩하기 위해서는 ASCII 불변 세트를 사용하거나, 실제 파일 시스템에 저장되어 있는 16진수 값에 대한 경로를 제공하십시오.

복수 바이트 환경에서, 불변 문자 세트에 속하지 않는 특수한 문자로 간주되는 네 개의 문자가 있습니다. 이 문자들은 다음과 같습니다.

- LIKE 처리에서 사용되는 2바이트 퍼센트 기호와 2바이트 밑줄 문자. LIKE에 대한 자세한 내용은 *SQL 참조서* 책을 참조하십시오.
- 그래픽 문자열의 공백 패딩에 사용되는 2바이트 공백 문자.
- 소스 코드 페이지와 목표 코드 페이지 사이에 맵핑이 존재하지 않을 때, 문자 변환 중에 대신 사용되는 2바이트 대체 문자.

코드 페이지별 이런 문자에 대한 코드 포인트는 다음과 같습니다.

표 19. 2바이트 특수 문자에 대한 코드 포인트

코드 페이지	2바이트 퍼센트	2바이트 밑줄	2바이트 공백	2바이트 대체 문자
932	X'8193'	X'8151'	X'8140'	X'FCFC'
938	X'8193'	X'8151'	X'8140'	X'FCFC'
942	X'8193'	X'8151'	X'8140'	X'FCFC'
943	X'8193'	X'8151'	X'8140'	X'FCFC'
948	X'8193'	X'8151'	X'8140'	X'FCFC'
949	X'A3A5'	X'A3DF'	X'A1A1'	X'AFFE'
950	X'A248'	X'A1C4'	X'A140'	X'C8FE'
954	X'A1F3'	X'A1B2'	X'A1A1'	X'F4FE'
964	X'A2E8'	X'A2A5'	X'A1A1'	X'FDFE'
970	X'A3A5'	X'A3DF'	X'A1A1'	X'AFFE'
1381	X'A3A5'	X'A3DF'	X'A1A1'	X'FEFE'
1383	X'A3A5'	X'A3DF'	X'A1A1'	X'A1A1'
13488	X'FF05'	X'FF3F'	X'3000'	X'FFFD'

UCS-2 고려사항: UCS-2 데이터베이스의 경우, 그래픽 공백은 X'0020'으로, 이것은 CCSID 13488에 사용되는『2 바이트 공백』X'3000'과는 다릅니다. EUC 데이터베이스의 데이터와 UCS-2 데이터베이스의 데이터를 비교할 때 이 차이점을 고려해야 합니다. UCS-2 데이터베이스에서 ASCII 퍼센트와 ASCII 밑줄의 유니코드 표기가 패턴 대응에 사용됨에 주의하십시오. DBCS 퍼센트와 DBCS 밑줄에

는 UCS-2 데이터베이스에 대한 특별한 의미가 없습니다. DBCS 대체 문자는 필요에 따라 EUC의 SBCS가 아닌 문자를 대체하는 데 사용됩니다. 3 바이트 또는 4 바이트 대체 문자의 개념은 없습니다.

원격 저장 프로시듀어와 UDF 코딩

원격으로 수행되는 저장 프로시듀어를 코딩할 때, 다음과 같은 내용을 고려해야 합니다.

- 저장 프로시듀어의 데이터는 데이터베이스 코드 페이지에 있어야 합니다.
- 문자 데이터 유형을 갖는 SQLDA를 사용하여 저장 프로시듀어로 전달되거나 저장 프로시듀어로부터 전달되는 데이터는 실제 문자 데이터를 가지고 있어야 합니다. 클라이언트 응용프로그램 코드 페이지가 데이터베이스 코드 페이지와 다른 경우, 숫자 데이터와 데이터 구조는 문자 유형과 함께 전달될 수 없습니다. 이는 서버가 SQLDA의 모든 문자 데이터를 변환하기 때문입니다. 문자 변환을 피하기 위해서는, BLOB 데이터 유형을 사용하여 2진 문자열 형식으로 데이터를 정의하거나, FOR BIT DATA로 문자 데이터를 정의하여 데이터를 전달할 수 있습니다.

기본적으로 DB2 DARI 저장 프로시듀어와 UDF를 호출할 때, 이들은 데이터베이스의 자국어 환경과 일치하지 않을 수도 있는 기본 자국어 환경 하에서 수행됩니다. 결과적으로, C `wchar_t` 그래픽 호스트 변수와 함수와 같은 국가나 코드 페이지 고유 작업을 수행하면 예상대로 작동하지 않을 수도 있습니다. 가능하다면, 저장 프로시듀어나 UDF 호출시 올바른 환경이 초기화되었는지를 확인해야 합니다.

혼합 코드 페이지 환경에서 패키지 이름 고려사항

패키지 이름은 사용자가 PRECOMPILE PROGRAM 명령이나 API를 호출할 때 결정됩니다. 기본적으로 이들은 응용프로그램 소스 파일의 확장자를 뗀 처음 8바이트에 기초하여 생성되며 대문자로 변환됩니다. 선택적으로 이름은 명시적으로 정의할 수 있습니다. 패키지 이름이 어디에서 비롯된 것인지와 상관없이, 동일하지 않은 코드 페이지 환경에서 수행중인 경우, 사용자의 패키지 이름의 문자는 불변 문자 세트 내에 있어야 합니다. 그렇지 않은 경우, 패키지 이름 수정에 관련된 문제점이 발생할 수 있습니다. 데이터베이스 관리 프로그램이 응용프로그램의 패키지를 찾을 수 없거나, 클라이언트 도구가 사용자 패키지의 올바른 이름을 표시하지 못할 수 있습니다.

패키지 이름의 문자 중 어느 하나라도 데이터베이스 코드 페이지의 유효한 문자에 직접 맵핑되지 않으면, 문자 변환으로 인한 패키지 이름 수정이 발생하게 됩니다. 그러한 경우, 대체 문자가 변환되지 않는 문자를 대체합니다. 그러한 수정이 발생한 후, 응용프로그램 코드 페이지로 다시 변환될 때 패키지 이름은 원래의 패키지 이름과 일치하지 않을 수 있습니다. 이러한 상황이 바람직하지 않은 경우의 예로, 사용자가 패키지 목록을 나열하고 이에 대한 작업을 하기 위해 DB2 Database Director를 사용하는 경우입니다. 표시된 패키지 이름이 예측한 이름과 같지 않을 수 있습니다.

패키지 이름에서 변환 문제가 발생하는 것을 방지하려면, 응용프로그램과 데이터베이스 코드 페이지 모두에서 유효한 문자만을 사용하도록 해야 합니다.

사전 처리 컴파일 및 바인딩

사전 처리 컴파일/바인드 시에 사전 처리 컴파일러는 응용프로그램을 실행합니다. 사전 처리 컴파일 요청 이전에 데이터베이스 연결이 이루어졌을 때 사용중인 코드 페이지가 사전 처리된 명령문 및 SQLCA에 리턴된 문자 데이터에 사용됩니다.

응용프로그램 실행

응용프로그램 실행시, 데이터베이스 연결이 이루어질 때 사용중인 사용자 응용프로그램의 코드 페이지가 연결되는 동안에 유효합니다. 모든 데이터는 이 코드 페이지에 기초하여 해석됩니다. 여기에는 동적 SQL문, 사용자 입력 데이터, 사용자 출력 데이터 및 SQLCA의 문자 필드가 포함됩니다.

주의사항

이 지침을 따르지 않을 경우 예상치 못한 결과가 발생할 수 있습니다. 이 상황은 데이터베이스 관리 프로그램이 탐지할 수 없으므로, 오류나 경고 메시지가 발생하지 않습니다. 예를 들어 C 응용프로그램에는 한 컬럼이 C1 CHAR(20)으로 정의된 테이블 T1에 대하여 수행되는 다음과 같은 SQL문이 포함되어 있습니다.

```
    (0)  EXEC SQL CONNECT TO GLOBALDB;
    (1)  EXEC SQL INSERT INTO T1 VALUES ('a-constant');
         strcpy(sqlstmt, "SELECT C1 FROM T1 WHERE C1='a-constant');
    (2)  EXEC SQL PREPARE S1 FROM :sqlstmt;
```

여기서,

x = 바인드시 응용프로그램 코드 페이지
y = 실행시 응용프로그램 코드 페이지
z = 데이터베이스 코드 페이지

바인드시 명령문 (1)의 '*a-constant*'가 코드 페이지 **x**에서 코드 페이지 **z**로 변환됩니다. 이 변환은 (x→z)로 표시할 수 있습니다.

실행시 '*a-constant*'(x→z)는 명령문 (1)이 실행될 때 테이블에 삽입됩니다. 그러나 명령문 (2)의 WHERE절은 '*a-constant*'(y→z)와 함께 실행됩니다. 상수의 코드 포인트에 있어 두 가지 변환(x→z 및 y→z)이 서로 다른 결과를 낳은 경우, 명령문 (2)의 SELECT는 명령문 (1)에 의해 삽입된 데이터를 검색하는 데 실패합니다.

다른 코드 페이지 간의 변환

이상적으로는 최적의 성능을 위해 사용자의 응용프로그램이 항상 데이터베이스와 같은 코드 페이지를 사용하는 것이 좋습니다. 그러나 이것이 항상 실제적이거나 가능한 것은 아닙니다. DB2 제품에서는 사용자의 응용프로그램과 데이터베이스가 서로 다른 코드 페이지를 사용하도록 허용하는 문자 변환을 지원합니다. 데이터의 의미를 유지하기 위해서는 하나의 코드 페이지의 문자가 다른 코드 페이지에 맵핑되어야 합니다.

문자 변환이 발생하는 시점: 문자 변환은 다음과 같은 상황에서 발생할 수 있습니다.

- 데이터베이스에 액세스하는 클라이언트나 응용프로그램이 데이터베이스의 코드 페이지와 다른 코드 페이지에서 수행중인 경우.

  이 데이터베이스 변환은 응용프로그램 코드 페이지에서 데이터베이스 코드 페이지로의 변환 및 데이터베이스 코드 페이지에서 응용프로그램 코드 페이지로의 변환 모두의 경우 데이터베이스 서버 머신에서 발생합니다.

  사용자는 특정 경우에 있어서는 클라이언트/서버 문자 변환을 최소화하거나 없애야 합니다. 예를 들면,

  – 주로 코드 페이지 850을 사용하는 OS/2와 Windows 클라이언트 응용프로그램과 일치하기 위해 코드 페이지 850을 사용하는 Windows NT 데이터베이스를 작성하십시오.

    Windows ODBC 응용프로그램이 Windows 데이터베이스 클라이언트에서 IBM DB2 ODBC 드라이버와 함께 사용된 경우, 이 문제점은 `odbc.ini`나 `db2cli.ini` 파일에서 TRANSLATEDLL 및 TRANSLATEOPTION 키워드를 사용함으로써 피할 수 있습니다.

- 주로 코드 페이지 850을 사용하는 OS/2와 DOS 클라이언트 응용프로그램 환경과 일치하기 위해, 코드 페이지 850을 사용하는 AIX용 DB2 데이터베이스를 작성하십시오.

주: OS/2용 DB2 버전 1.0 또는 버전 1.2 데이터베이스 서버는 다른 코드 페이지 간의 문자 변환을 지원하지 않습니다. 서버와 클라이언트의 코드 페이지가 호환되는지 확인하십시오. 지원되는 코드 페이지 변환 목록에 대해 *관리 안내서*를 참조하십시오.

- PC/IXF 파일을 가져오는 클라이언트나 응용프로그램이 가져오기를 하고 있는 파일과 다른 코드 페이지에서 수행되는 경우.

  이 데이터 변환은 클라이언트가 데이터베이스 서버에 액세스하기 전에 데이터베이스 클라이언트 머신에서 발생합니다. 응용프로그램이 이전 시점에서 앞서 언급한 데이터베이스의 코드 페이지와 다른 코드 페이지에서 수행되는 경우, 추가적인 데이터 변환이 발생할 수 있습니다.

  데이터 변환이 있는 경우, 이것은 가져오기 유틸리티가 호출되는 방식에 따라 다릅니다. 자세한 내용은 *관리 안내서* 책을 참조하십시오.

- 호스트 또는 AS/400 서버의 데이터에 액세스하는 데 DB2 Connect가 사용된 경우. 이 경우 데이터를 수신하는 쪽이 문자 데이터를 변환합니다. 예를 들어 MVS/ESA용 DB2로 송신된 데이터는 MVS/ESA용 DB2에 의해 적절한 MVS 코드화된 문자 세트 식별자(CCSID)로 변환됩니다. MVS/ESA용 DB2에서 DB2 Connect 머신으로 다시 송신된 데이터는 DB2 Connect에 의해 변환됩니다. 자세한 정보는 *DB2 Connect 사용자 안내서* 책을 참조하십시오.

다음에 대해서는 문자 변환이 발생하지 **않습니다**.

- 파일 이름. 파일 이름의 경우 ASCII 불변 세트를 사용하거나, 실제로 파일 시스템에 저장되는 16진수 값으로 파일 이름을 지정해야 합니다. SQL문의 일부에 파일 이름을 포함하는 경우, 명령문 변환의 일부로 변환됨에 주의하십시오.
- 목표가 되는 데이터 또는 FOR BIT DATA 속성이 할당된 컬럼으로부터 온 데이터 또는 그 결과가 FOR BIT이거나 BLOB 데이터인 SQL 조작에 사용된 데이터. 이러한 경우, 데이터는 바이트 스트림으로 취급되고, 변환은 일어나지 않

습니다. [1] 문자열 지정, 비교 및 조합을 위한 동일하지 않은 코드 페이지 규칙에 대한 정보는 *SQL 참조서* 책을 참조하십시오.

- 원하는 코드 페이지 조합을 지원하지 않거나 그에 대한 지원이 설치되어 있지 않은 DB2 제품 또는 플랫폼. 이 경우, 사용자가 응용프로그램을 수행하려고 할 때 SQLCODE -332(SQLSTATE 57017)가 리턴됩니다.

변환 중 문자 대체: 사용자의 응용프로그램이 하나의 코드 페이지에서 다른 코드 페이지로 변환할 때, 하나 이상의 문자가 목표 코드 페이지에서 표시되지 않을 수 있습니다. 이러한 경우, DB2는 표시되지 않은 문자 대신 목표 문자열에 *대체* 문자를 삽입합니다. 그러면 대체 문자는 문자열의 유효한 일부로 간주됩니다. 대체가 발생하는 경우, SQLCA의 SQLWARN10 표시기가 'W'로 설정됩니다.

주: WCHARTYPE CONVERT 사전 처리 컴파일러 옵션을 사용하여 발생하는 문자 변환의 경우 대체가 발생하는 경우 경고를 플래그하지 않습니다.

지원되는 문자 변환: 데이터 변환이 발생할 때, 변환은 **소스 코드 페이지**에서 **목표 코드 페이지**로 이루어집니다.

소스 코드 페이지는 데이터의 소스로부터 결정됩니다. 응용프로그램의 데이터는 응용프로그램 코드 페이지와 동일한 소스 코드 페이지를 갖고, 데이터베이스의 데이터는 데이터베이스 코드 페이지와 동일한 소스 코드 페이지를 갖습니다.

목표 코드 페이지의 결정은 더 많은 것과 관련이 됩니다. 중간 조작 규칙을 포함하여 해당 데이터가 어디에 위치할 것인가 등이 고려됩니다.

- 데이터가 중간 조작 없이 응용프로그램에서 데이터베이스로 직접 이동하는 경우, 목표 코드 페이지는 데이터베이스 코드 페이지입니다.
- 데이터가 PC/IXF 파일의 데이터베이스로 가져오기 되는 경우, 두 단계의 문자 변환 단계를 수행합니다.
  1. PC/IXF 파일 코드 페이지(소스 코드 페이지)에서 응용프로그램 코드 페이지(목표 코드 페이지)로.
  2. 응용프로그램 코드 페이지(소스 코드 페이지)에서 데이터베이스 코드 페이지(목표 코드 페이지)로.

---

1. 그러나 FOR BIT DATA로 정의된 컬럼에 삽입된 리터럴은 이 리터럴이 변환된 SQL문의 일부인 경우 변환됩니다.

두 가지 변환 단계가 발생하는 경우 주의해야 합니다. 문자 데이터 유실을 방지하려면 *관리 안내서*에 나열된 지원되는 문자 변환을 반드시 따르십시오. 또한 각 그룹 내에서, 소스와 목표 코드 페이지 모두에 존재하는 문자들만이 의미있는 변환을 할 수 있습니다. 다른 문자들은 『대체』로 사용되고, 목표 코드 페이지에서 다시 소스 코드 페이지로 변화할 때에만 유용하며, 위에서 설명한 두 단계의 변환 프로세스에서 반드시 의미 없는 변환을 제공한다고는 할 수 없습니다. 응용프로그램 코드 페이지가 데이터베이스 코드 페이지와 동일하다면, 이러한 문제점은 방지할 수 있습니다.

- 소스가 응용프로그램 코드 페이지, 데이터베이스 코드 페이지, FOR BIT DATA 또는 BLOB 데이터 중 어느 하나인 문자 데이터에 대하여 수행된 작업으로부터 데이터를 구하는 경우, 데이터 변환은 일련의 규칙에 근거하여 이루어집니다. 최종 목표 코드 페이지가 판별되기 전에 데이터 항목의 일부 또는 모두가 중간 결과로 변환되어야 합니다. 이런 규칙에 대한 요약과 개별 연산자 및 술어가 있는 특정 응용프로그램에 대한 내용은 *SQL 참조서*를 참조하십시오.

DB2 Universal Database에서 지원되는 코드 페이지 목록에 대해 *관리 안내서*를 참조하십시오. 『그룹』 제목 아래의 값들은 어디에서 변환이 지원되는지를 판별하는 데 사용될 수 있습니다. 어떤 코드 페이지든 같은 IBM 정의 언어 그룹에 나열된 다른 코드 페이지로 변환될 수 있습니다. 예를 들어, 코드 페이지 437은 37, 819, 850, 1051, 1252 또는 1275로 변환될 수 있습니다.

주: 예를 들어, DBCS와 EUC와 같은 복수 바이트 코드 페이지 간의 문자열 변환의 경우, 문자열의 길이가 길어지거나 짧아질 수 있습니다.

문자 변환 확장 인수: 사용자 응용프로그램이 DB2 데이터베이스 서버로의 연결 시도를 완료하면, 리턴된 SQLCA의 다음과 같은 필드를 고려해야 합니다.

- SQLERRMC 필드의 두번째 토큰(토큰은 X'FF'로 구분됩니다)은 데이터베이스의 코드 페이지를 나타냅니다. SQLERRMC 필드의 아홉번째 토큰은 응용프로그램의 코드 페이지를 나타냅니다. 응용프로그램의 코드 페이지를 조회하고 이것을 데이터베이스의 코드 페이지에 비교하는 것은 응용프로그램에게 문자 변환이 수행될 연결이 설정되었는지 여부를 알려주는 것입니다.
- SQLERRD 배열의 첫번째와 두번째 항목. SQLERRD(1)에는 응용프로그램 코드 페이지에서 데이터베이스 코드 페이지로 변환될 때, 혼합 문자 데이터(CHAR

데이터 유형)의 길이에 대한 최대 예상 확장 또는 축소 인수와 동일한 정수 값이 들어 있습니다. SQLERRD(2)에는 데이터베이스 코드 페이지에서 응용프로그램 코드 페이지로 변환될 때, 혼합 문자 데이터(CHAR 데이터 유형)의 길이에 대한 최대 예상 확장 또는 축소 인수와 동일한 정수 값이 들어 있습니다. 0이나 1값은 확장하지 않음을 의미합니다. 1보다 큰 값은 길이에서 확장이 가능함을 나타내며, 음수 값은 축소가 가능함을 나타냅니다. CONNECT문 사용에 대한 세부사항은 *SQL 참조서* 책을 참조하십시오.

그래픽 문자열 데이터에 대한 고려사항이 동일하지 않은 코드 페이지 상황에서 인수가 되어서는 안됩니다. 각 문자열은 데이터가 응용프로그램 코드 페이지에 있는지, 데이터베이스 코드 페이지에 있는지 여부와 상관없이 같은 수의 문자를 갖습니다.

동일하지 않은 코드 페이지 상황 처리에 대한 정보는 587 페이지의 『동일하지 않은 코드 페이지 상황』의 내용을 참조하십시오.

DBCS 문자 집합

결합된 1바이트 문자 세트(SBCS)나 2바이트 문자 세트(DBCS) 코드 페이지에서 1바이트와 2바이트 문자 코드 포인트를 사용할 수 있습니다. 이는 보통 코드 포인트의 나머지를 정의하지 않거나 2바이트 코드 포인트의 첫번째 바이트에 할당하여 1바이트 문자의 혼합 코드 테이블에서 사용할 수 있는 256 코드 포인트의 부속 집합을 예약하여 이루어집니다. 이런 코드 포인트는 다음 테이블에 있습니다.

표 20. 혼합 문자 세트 코드 포인트

국가	혼합 코드 페이지 지원	1바이트 문자의 코드 포인트	2바이트 문자의 첫번째 바이트에 대한 코드 포인트
일본	932, 943	x00-7F, xA1-DF	x81-9F, xE0-FC
일본	942	x00-80, xA0-DF, xFD-FF	x81-9F, xE0-FC
타이완	938 (*)	x00-7E	x81-FC
타이완	948 (*)	x00-80, FD, FE	x81-FC
한국	949	x00-7F	x8F-FE
타이완	950	x00-7E	x81-FE
중국	1381	x00-7F	x8C-FE
한국	1363	x00-7F	x81-FE
중국	1386	x00	x81-FE
주: (*) 이는 더 이상 권장되지 않는 이전 코드 페이지입니다.			

이런 범주에 할당되지 않은 코드 포인트는 정의되지 않고 1바이트의 정의되지 않은 코드 포인트로 처리됩니다.

내재된 DBCS 코드 테이블에는 유효한 첫번째 바이트의 둘째 바이트로 사용할 수 있는 256 코드 포인트가 있습니다. 둘째 바이트 값에는 0x40에서 0x7E까지의 값과 0x80에서 0xFE까지의 값이 있을 수 있습니다. DBCS 환경에서 DB2는 개별 2바이트 문자에서 유효성 검사를 수행하지 않습니다.

확장 UNIX 코드(EUC) 문자 세트

EUC 코드 페이지에서 1바이트 문자 코드 포인트와 최대 3개의 서로 다른 다중 바이트 문자 코드 포인트를 사용할 수 있습니다. 이는 1바이트 문자의 내재된 SBCS 코드 페이지 식별자에 대해 사용할 수 있는 256 코드 포인트의 부속 집합을 예약하여 이루어집니다. 코드 포인트의 나머지는 정의되지 않고 다중 바이트 문자의 요소로 할당되거마 다중 바이트 문자의 단일 이동 도입자로 할당됩니다. 이런 코드 포인트는 다음 테이블에 있습니다.

표 21. 일본어 EUC 코드 포인트

그룹	첫째 바이트	둘째 바이트	셋째 바이트	네째 바이트
G0	x20-7E	n/a	n/a	n/a
G1	xA1-FE	xA1-FE	n/a	n/a
G2	x8E	xA1-FE	n/a	n/a
G3	x8E	xA1-FE	xA1-FE	n/a

표 22. 한국어 EUC 코드 포인트

그룹	첫째 바이트	둘째 바이트	셋째 바이트	네째 바이트
G0	x20-7E	n/a	n/a	n/a
G1	xA1-FE	xA1-FE	n/a	n/a
G2	n/a	n/a	n/a	n/a
G3	n/a	n/a	n/a	n/a

표 23. 대만어 EUC 코드 포인트

그룹	첫째 바이트	둘째 바이트	셋째 바이트	네째 바이트
G0	x20-7E	n/a	n/a	n/a
G1	xA1-FE	xA1-FE	n/a	n/a
G2	x8E	xA1-FE	xA1-FE	xA1-FE
G3	n/a	n/a	n/a	n/a

표 24. 중국어 EUC 코드 포인트

그룹	첫째 바이트	둘째 바이트	셋째 바이트	네째 바이트
G0	x20-7E	n/a	n/a	n/a
G1	xA1-FE	xA1-FE	n/a	n/a
G2	n/a	n/a	n/a	n/a
G3	n/a	n/a	n/a	n/a

이런 범주에 할당되지 않은 코드 포인트는 정의되지 않고 1바이트의 정의되지 않은 코드 포인트로 처리됩니다.

DBCS 환경에서 CLI/ODBC/JDBC/SQLJ 프로그램 수행

2 바이트 문자 세트(DBCS) 환경에서 DB2 Universal Database에 액세스하는 Java 프로그램 수행에 대한 세부사항은 DB2 Java - DBCS 지원 온라인

(http://www.ibm.com/software/data/db2/java/dbcsjava.html)을 참조하십시오. 현재 이 웹 페이지에는 다음과 같은 정보가 들어 있습니다.

JDBC와 SQLJ 프로그램은 DB2 CLI/ODBC 드라이버를 사용하여 DB2에 액세스하므로, 같은 구성 파일(db2cli.ini)을 사용합니다. DBCS 환경에서 DB2 Universal Database에 액세스하는 Java 프로그램을 수행하는 경우, 이 구성 파일에 다음과 같은 항목들이 추가되어야 합니다.

PATCH1 = 65536

이것은 실제로 그래픽 리터럴인 문자 리터럴 앞에 드라이버가 수동으로 "G"를 강제로 삽입합니다. 이 PATCH1 값은 2 바이트 환경에서 작업할 때는 항상 설정되어야 합니다.

PATCH1 = 64

이것은 드라이버가 그래픽 출력 문자열을 널(NULL)로 강제로 종료합니다. 이것은 2 바이트 환경의 Microsoft Access에서 필요합니다. 이 PATCH1 값 역시 사용할 필요가 있는 경우, 두 값을 더하여(64+65536+65600) PATCH1=65600으로 설정할 수 있습니다. PATCH1 값을 여러개 지정하는 것에 관한 정보는 아래의 주 #2를 참조하십시오.

PATCH2 = 7

이것은 드라이버가 강제로 char 컬럼 데이터 유형으로 모든 그래픽 컬럼 데이터 유형을 맵핑하도록 합니다. 이것은 2 바이트 환경에서 필요합니다.

PATCH2 = 10

이 설정은 EUC(확장 Unix 코드) 환경에서만 사용되어야 합니다. 이것은 JDBC에 적절한 형식으로 CLI 드라이버가 CHAR, VARCHAR 등과 같은 문자 변수를 제공하도록 합니다. 이러한 문자 유형의 데이터는 이 설정 없이는 JDBC에서 사용할 수 없습니다.

주:

1. 이 키워드들 각각은 db2cli.ini 파일의 각 데이터베이스 고유의 스탠자에 설정됩니다. 여러 데이터베이스에 대하여 설정하려면, db2cli.ini의 각 데이터베이스 스탠자에 이들을 반복해서 설정해야 합니다.
2. 여러 PATCH1 값을 설정하려면, 각각 개별적인 값을 더하여 그 합을 사용합니다. PATCH1을 64와 65536으로 설정하려면, PATCH1=65600

(64+65536)을 설정합니다. 이미 다른 PATCH1 값이 설정되어 있다면, 기존 숫자를 기존의 숫자와 추가하려는 새로운 PATCH1 값을 더한 값으로 대체하십시오.

3. PATCH2 값을 여러개 설정하려면, PATCH1 옵션과는 달리 쉼표로 구분한 문자열로 지정하십시오. PATCH2 값을 1과 7로 설정하려면, PATCH2="1,7"로 설정합니다.

이들 키워드 설정에 대한 자세한 정보는 *설치 및 구성 보충 설명서* 책을 참조하십시오.

일본어 및 대만어 EUC 및 UCS-2 코드 세트 변환

확장 UNIX 코드(EUC)는 UNIX를 기초로 한 운영 환경에서 1개에서 4개까지의 문자 세트로부터 지원하는 일반적인 코드화 규칙 세트를 말합니다. 코드화 규칙은 문자 세트의 일부를 구별하는 데 제어 문자가 사용되는 7 비트와 8 비트 데이터 코드화를 위한 ISO 2022 정의에 기초합니다. EUC는 코드 세트 코드화 체계라기보다는 코드 세트의 콜렉션을 지정하는 수단입니다. EUC에 기초한 코드 세트는 EUC 코드화 규칙을 따르지만, 특정 인스턴스와 연관된 특정 문자 세트를 식별하기도 합니다. 예를 들어, 일본어에 대한 IBM-eucJP 코드 세트는 EUC 코드화 규칙에 따른 일본어 산업 표준의 코드화를 말합니다. 지원되는 코드 페이지 목록은 사용자가 사용하고 있는 플랫폼의 *빠른 시작* 책을 참조하십시오.

길이가 2 바이트보다 큰 문자 코드화를 사용하는 EUC 코드 페이지 하에 수행되는 동안, 그래픽(순수한 2 바이트 문자) 데이터에 대한 데이터베이스와 클라이언트 응용프로그램 지원은 제한됩니다. DB2 Universal Database 제품은 그래픽 데이터에 대하여 모든 문자가 정확히 2 바이트여야 한다는 엄격한 규칙을 부여합니다. 이 규칙으로 인해 일본어와 대만어 EUC 코드 페이지의 많은 문자들이 허용되지 않습니다. 이러한 상황을 극복하기 위해, 다른 코드화 체계를 사용하여 일본어 및 대만어 EUC 그래픽 데이터를 나타내도록, 응용프로그램 레벨과 데이터베이스 레벨 모두에서의 지원이 제공됩니다.

일본어나 대만어 EUC 코드 페이지에서 작성된 데이터베이스는 완전한 ISO 10646 표준의 적절한 부속 집합으로 2 바이트 코드화 체계인 ISO 10646 UCS-2 코드 세트를 사용하여 그래픽 데이터를 저장하고 조작합니다. 마찬가지로, 이들 코드 페이지 하에서 수행되는 응용프로그램은 UCS-2 코드화 데이터로 그래픽 데이터를 데

이터베이스 서버로 송신합니다. 이러한 지원을 사용하여 EUC 코드 페이지 하에서 수행되는 응용프로그램은 DBCS 코드 페이지 하에서 수행되는 응용프로그램과 동일한 데이터 유형에 액세스할 수 있습니다. EUC 환경에 관한 추가 정보는 *SQL 참조서* 책을 참조하십시오. UCS-2와 연관된 IBM 정의 코드 페이지 식별자는 1200이며, 같은 코드 페이지의 CCSID 번호는 13488입니다. eucJP나 eucTW 데이터베이스의 그래픽 데이터는 CCSID 번호 13488을 사용합니다. UCS-2 데이터베이스에서는 GRAPHIC 데이터에 코드 페이지 번호 1200을 사용하십시오.

ISO 10646 표준은 인도어, 타이어, 아라비아어 및 히브리어와 같은 몇 개의 스크립트에서 필요한 *조합* 문자들의 코드화를 지정합니다. 또한 이 문자들은 라틴, 키릴 및 그리스어 스크립트의 생성에도 사용할 수 있습니다. 그러나 이들의 존재는 같은 텍스트에 대한 대체 코딩을 가능케 합니다. 코딩이 분명하고 데이터 무결성이 유지된다고 하더라도, 조합 문자가 들어 있는 텍스트 처리는 더 복잡합니다. 조합 문자를 처리하지 않기로 한 응용프로그램에 따르기 위해 ISO 10646은 세 가지 구현 레벨을 정의합니다.

Level 1.
조합 문자를 허용하지 않습니다.

Level 2.
조합 문자의 부속 집합만 허용됩니다.

Level 3.
모든 조합 문자가 허용됩니다.

DB2 Universal Database는 모든 조합 문자를 포함하는 UCS-2 문자 세트 전체를 지원하지만, 문자의 조합 또는 해체를 수행하지는 않습니다. 유니코드 표준에 대한 자세한 정보는 Addison-Wesley 출판사의 *유니코드 표준 버전 2.0*을 참조하십시오. UCS-2에 대한 자세한 정보는 국제 표준 기구의 *ISO/IEC 10646-1*을 참조하십시오.

이 문자 세트를 사용하는 응용프로그램이나 데이터베이스에 대해 작업하는 경우, UCS-2 코드화 데이터 처리를 고려해야 할 것입니다. UCS-2 그래픽 데이터를 응용프로그램의 EUC 코드 페이지로 변환할 때, 데이터 길이가 길어질 가능성이 있습니다. 데이터 확장에 대한 세부사항은 576 페이지의 『문자 변환 확장 인수』의

내용을 참조하십시오. 대용량의 데이터가 표시될 때, 버퍼를 할당하고 일련의 프래그먼트에서 데이터를 변환 및 표시할 필요가 있습니다.

다음 절에서는 이 환경에서 데이터를 처리하는 방법에 대하여 설명합니다. 이 절에서 EUC라는 용어는 일본어와 대만어 EUC 문자 세트만을 의미합니다. 이러한 논의는 DB2 한글이나 중국어 EUC 지원에는 적용되지 않는데, 이들 문자 세트의 그래픽 데이터는 EUC 코드화를 사용하여 나타내기 때문입니다.

혼합 EUC 및 2 바이트 클라이언트 및 데이터베이스 고려사항

혼합 EUC 및 2 바이트 코드 페이지 환경에서의 데이터베이스 오브젝트는 클라이언트와 데이터베이스 코드 페이지간의 변환 결과로 오브젝트 이름의 길이가 늘어나거나 줄어들 수 있는 가능성으로 인해 복잡합니다. 특히 다수의 관리 명령들 및 유틸리티들에서는 그들이 입력이나 출력 매개변수로 받아들일 수 있는 문자열의 길이에 제한을 두고 있습니다. 이러한 제한은 달리 문서화되어 있지 않은 한 보통 클라이언트에서 적용됩니다. 예를 들어 테이블 이름의 길이 제한은 128 바이트입니다. 2 바이트 코드 페이지에서 128 바이트인 문자 세트는 EUC 코드 페이지에서는 135 바이트입니다. 이러한 135 바이트의 테이블 이름은 목표 2 바이트 데이터베이스에서는 유효하더라도 입력 매개변수로 사용되는 경우 REORGANIZE TABLE과 같은 명령은 유효하지 않은 것으로 간주합니다. 마찬가지로 출력 매개변수로 허용되는 최대 길이는 데이터베이스 코드 페이지에서 응용프로그램 코드 페이지로의 변환 후에 초과할 수 있습니다. 이것은 변환 오류 또는 출력 데이터 절단이 발생하는 원인이 될 수도 있습니다.

혼합 EUC 및 2 바이트 환경에서 관리 명령과 유틸리티들을 많이 사용할 것으로 예측되는 경우, 데이터베이스 오브젝트와 연관된 데이터를 정의할 때, 지원되는 한계를 넘어 길이가 확장될 수 있는 가능성을 염두에 두어야 합니다. 2 바이트 클라이언트에서 EUC 데이터베이스를 관리하는 경우, EUC 클라이언트에서 2 바이트 데이터베이스를 관리하는 것보다 제한 사항이 훨씬 적습니다. 2 바이트 문자열은 항상 길이에 있어서 해당 EUC 문자열과 같거나 더 짧습니다. 그러면 문자열 길이 한계에 의해 발생할 수 있는 문제점이 일반적으로 훨씬 적습니다.

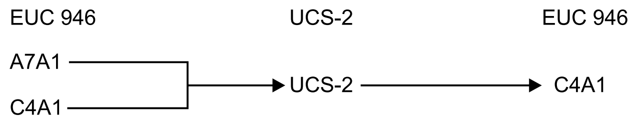
주: SQL문의 경우, 입력 매개변수의 검증은 전체 명령문이 데이터베이스 코드 페이지로 변환될 때까지 수행되지 않습니다. 이렇게 사용자는 클라이언트 코드 페

이지로 표시될 때 허용되는 것보다 기술적으로 더 길지만, 데이터베이스 코드 페이지에서 표시될 때는 길이 요구사항을 충족시키는 문자열을 사용할 수 있습니다.

대만어 사용자들을 위한 고려사항

대만어의 표준 정의로 인해, 2 바이트나 EUC 코드 페이지와 UCS-2 간의 일부 문자를 변환할 때 부가작용이 발생할 수 있습니다. 이 코드 세트의 다른 문자로 변환될 때, 같은 UCS-2 코드 포인트를 공유하는 189개의 문자(187개의 어근과 2개의 숫자로 구성)가 있습니다. 이 문자들이 다시 2 바이트나 EUC로 변환될 때, 이들은 원래의 코드 포인트로 변환되지 않고, 동일한 UCS-2 코드 포인트를 공유하는 동일한 문자의 표의문자(ideograph) 코드 포인트로 변환됩니다. 표시되면, 문자는 동일하게 보이지만 코드 포인트가 다릅니다. 응용프로그램의 설계에 따라 이러한 기능을 고려해야 할 수도 있습니다.

예를 들어, EUC 코드 페이지 964가 UCS-2로 변환되었다가 다시 원래의 코드 페이지 EUC 946으로 변환된 경우, 코드 포인트 A7A1가 어떻게 될 것인지를 생각해 보십시오.



이렇게 원래의 코드 포인트 A7A1과 C4A1은 변환 후에 코드 포인트 C4A1이 됩니다.

EUC 코드 페이지 946(대만어 EUC)이나 950(대만어 Big-5) 및 UCS-2의 코드 페이지 변환 테이블이 필요한 경우, 온라인 제품 및 서비스 기술 라이브러리 (http://www.ibm.com/software/data/db2/library/)를 참조하십시오.

일본어 또는 대만어 EUC 응용프로그램 개발

EUC 응용프로그램을 개발할 때 다음과 같은 항목을 고려해야 합니다.

- 그래픽 데이터 처리
- 혼합 코드 세트 환경에서의 개발

저장 프로시듀어에 대한 추가적인 고려사항은 586 페이지의 『저장 프로시듀어에 대한 고려사항』의 내용을 참조하십시오. 추가적인 고유 언어별 응용프로그램 개발에 관한 정보는 다음에서 설명됩니다.

- 700 페이지의 『C 및 C++의 일본어 또는 대만어 EUC 및 UCS-2 고려사항』(C 및 C++).
- 785 페이지의 『COBOL에 대한 일본어 또는 대만어 EUC 및 UCS-2 고려사항』(COBOL).
- 803 페이지의 『FORTRAN에 대한 일본어 또는 대만어 EUC 및 UCS-2 고려사항』(FORTRAN).
- 826 페이지의 『REXX에 대한 일본어 또는 대만어 EUC 고려사항』(REXX).

그래픽 데이터 처리: 이 절에서는 그래픽 데이터를 처리하기 위한 EUC 응용프로그램 개발 고려사항에 대하여 설명합니다. 여기에는 그래픽 상수 처리, UDF에서의 그래픽 데이터 처리, 저장 프로시듀어, DBCLOB 파일 및 조합 등이 포함됩니다.

그래픽 상수: 그래픽 상수 또는 리터럴은 실제 SQL문의 일부로서 혼합 문자 데이터로 분류됩니다. 일본어나 대만어 EUC 클라이언트로부터의 SQL문의 그래픽 상수는 내재적으로 데이터베이스 서버에 의해 그래픽 코드화로 변환됩니다. SQL 응용프로그램에서 EUC 코드화 문자로 구성된 그래픽 리터럴을 사용할 수 있습니다. EUC 데이터베이스 서버는 UCS-2가 될 그래픽 데이터베이스 코드 세트로 이러한 리터럴들을 변환합니다. EUC 클라이언트의 그래픽 상수에는 CS0 7 비트 ASCII 문자나 일본어 EUC CS2(Katakana) 문자와 같은 단일 폭의 문자가 들어가서는 안됩니다.

그래픽 상수에 대한 추가 정보는, *SQL 참조서* 책을 참조하십시오.

UDF에 대한 고려사항: UDF는 데이터베이스 서버에서 호출되고, 데이터베이스와 같은 코드 세트로 코드화되는 데이터를 처리하도록 합니다. 일본어나 대만어 코드 세트에서 수행되는 데이터베이스의 경우, 데이터베이스가 작성되는 EUC 코드 세트를 사용하여 혼합 문자 데이터가 코드화됩니다. 그래픽 데이터는 UCS-2를 사용하여 코드화됩니다. 이것은 UDF가 UCS-2를 사용하여 코드화되는 그래픽 데이터를 인식하고 처리해야 함을 의미합니다.

예를 들어, 그래픽 문자열을 혼합 문자열로 변환하는 VARCHAR이라고 하는 UDF를 작성합니다. 데이터베이스가 EUC 코드 세트에서 작성된 경우, VARCHAR 함수는 UCS-2로 코드화된 그래픽 문자열을 변환해야 합니다.

저장 프로시듀어에 대한 고려사항: 일본어나 대만어 EUC 코드 세트 중 하나에서 수행되는 저장 프로시듀어는 UCS-2를 사용하여 코드화된 그래픽이 데이터를 인식하고 처리하기 위한 준비를 해야 합니다. 이러한 코드 세트를 수행할 때, 저장 프로시듀어의 입/출력 SQLDA를 통해 수신되고 리턴된 그래픽 데이터는 UCS-2를 사용하여 코드화됩니다.

DBCLOB 파일에 대한 고려사항: DBCLOB 파일에는 두 가지 중요한 고려사항이 있습니다.

- DBCLOB 파일 데이터는 응용프로그램의 EUC 코드 페이지에 있는 것으로 간주됩니다. EUC DBCLOB 파일의 경우, 데이터는 읽기를 수행할 때는 클라이언트의 UCS-2로 변환되고, 쓰기를 수행할 때에는 UCS-2로부터 변환됩니다.
- 서버에서 읽혀지거나 씌어진 바이트 수는 파일로부터 읽혀졌거나 파일로 씌어진 UCS-2 코드화 문자의 수에 근거한 파일 참조 변수의 데이터 길이 필드로 리턴됩니다. 파일로부터 읽혀지거나 파일에 기록되는 실제 바이트 수는 더 클 수 있습니다.

조합: 그래픽 데이터는 2진 순서로 정렬됩니다. 혼합 데이터는 각 바이트에 적용된 데이터베이스의 조합 순서에 정렬됩니다. 정렬 순서에 대한 정보는 *SQL 참조서* 책을 참조하십시오. 같은 국가에서도 EUC 코드 세트와 DBCS 코드 세트에서의 문자 순서에서의 차이로 인해, 같은 데이터가 EUC 데이터베이스와 DBCS 데이터베이스에 정렬될 때 다른 결과가 얻어질 수 있습니다.

혼합 코드 세트 환경에서의 개발

이 절에서는 혼합 EUC 및 DBCS 환경에서 응용프로그램을 개발할 때 특정 상황에서 데이터 길이의 증가나 감소와 관련된 다음과 같은 고려사항을 설명합니다.

- 동일하지 않은 코드 페이지 상황
- 클라이언트에 기초한 매개변수 검증
- DESCRIBE문 사용
- 고정 또는 가변 길이 데이터 유형 사용
- 문자 변환 문자열 길이 오버플로우

- 유니코드(UCS-2) 데이터베이스에 연결된 응용프로그램
- 문자열 변환 규칙
- 데이터 유형 한계를 초과한 문자 변환
- 저장 프로시듀어에서의 문자 변환

동일하지 않은 코드 페이지 상황: 응용프로그램 코드 페이지와 데이터베이스 코드 페이지에 의해 사용되는 문자 코드화 체계에 따라, 소스 코드 페이지에서 목표 코드 페이지로 변환될 때 문자열의 길이가 변경될 수도, 되지 않을 수도 있습니다. 길이 변경은 보통 복수 바이트 코드 페이지와 다른 코드화 체계, 예를 들면 DBCS와 EUC간의 변환과 관련됩니다.

메모리를 낮게 할당하는 것보다 더 많이 할당하는 것의 문제가 훨씬 덜하므로, 길이가 줄어들 가능성보다 늘어날 가능성이 더 심각합니다. 이러한 확장이 일어날 가능성이 어디에 있느냐에 따라 데이터 송신 및 검색시 응용프로그램 고려사항은 따로 다루어져야 합니다. 길이의 증감이 예상되는 상황에서 *최선의 경우*와 *최악의 경우* 간의 차이에도 주의를 기울이는 것이 중요합니다. 확장 가능성을 나타내는 양수 값은 *최악의 경우* 곱하기 인수를 제공합니다. 예를 들어 SQLERRD(1)이나 SQLERRD(2)의 2 값은 저장영역의 문자열 길이의 최대 2배가 변환후 데이터를 처리하는 데 필요함을 의미합니다. 이것은 *최악의 경우* 표시기입니다. 이 예에서 *최선의 경우*는 변환 후에도 길이가 동일하게 남아 있는 것입니다.

감소 가능성을 나타내는 SQLERRD(1) 또는 SQLERRD(2)의 음수 값 역시 *최악의 경우* 확장 인수를 제공합니다. 예를 들어 -1 값은 필요한 최대 저장영역이 변환 전에 문자열의 길이와 같음을 의미합니다. 사실상 더 작은 저장영역이 필요할 수도 있지만, 실제적으로 수신하는 응용프로그램이 소스 데이터의 구조를 미리 알수 없는 한 거의 소용이 없습니다.

문자 변환 후 최대한 확장될 가능성을 고려하여 충분한 저장영역이 할당되도록 하려면, 다음과 같은 계산으로 산출된 `max_target_length` 값의 저장영역을 할당해야 합니다.

1. 데이터의 확장 인수를 판별하십시오.

   응용프로그램에서 데이터베이스로의 데이터 전송의 경우,

```
expansion_factor = ABS[SQLERRD(1)]
if expansion_factor = 0
   expansion_factor = 1
```

데이터베이스에서 응용프로그램으로의 데이터 전송의 경우,

```
expansion_factor = ABS[SQLERRD(2)]
if expansion_factor = 0
   expansion_factor = 1
```

위의 계산에서 ABS는 절대 값을 말합니다.

일부 DB2 Universal Database 제품은 SQLERRD(1)과 SQLERRD(2)에서 0을 리턴하므로 expansion_factor = 0을 확인하는 것이 필요합니다. 이러한 서버들은 데이터의 확장이나 감소를 발생시키는 코드 페이지 변환을 지원하지 않습니다. 이것은 확장 인수 1로 표시됩니다.

2. 중간 길이 계산.

```
temp_target_length = actual_source_length * expansion_factor
```

3. 목표 데이터 유형의 최대 길이를 판별하십시오.

목표 데이터 유형	유형의 최대 길이(**type_maximum_length**)
CHAR	254
VARCHAR	32 672
LONG VARCHAR	32 700
CLOB	2 147 483 647

4. 최대 목표 길이를 판별하십시오.

```
1   if temp_target_length < actual_source_length
       max_target_length = type_maximum_length
    else
2   if temp_target_length > type_maximum_length
       max_target_length = type_maximum_length
    else
3   max_target_length = temp_target_length
```

길이 계산시 발생할 수 있는 오버플로우를 허용하려면 위의 사항들을 모두 확인해야 합니다. 특정 확인은 다음과 같습니다.

1 단계 2의 temp_target_length 계산 중에 오버플로우가 발생합니다.

두 개의 양수 값을 곱한 결과가 데이터 유형의 최대 값보다 큰 경우, 결과는 랩되어(wrap around) 두 값 중 큰 값보다 작은 값으로 리턴됩니다.

예를 들어, 2 바이트의 부호 있는 정수(비 CLOB 데이터 유형 길이에 사용되는)는 32 767입니다. actual_source_length가 25 000이고, 확장 인수가 2인 경우, temp_target_length는 이론적으로 50 000입니다. 이 값은 2 바이트 부호 있는 정수에는 너무 크므로, 랩되어 -15 536으로 리턴됩니다.

CLOB 데이터 유형의 경우, 4 바이트 부호 있는 정수는 길이에 사용됩니다. 4 바이트 부호 있는 정수의 최대 값은 2 147 483 647입니다.

2 temp_target_length는 이 데이터 유형에 너무 큽니다.

데이터 유형의 길이는 단계 3에서 나열된 값을 초과할 수 없습니다.

변환시 데이터 유형에 사용할 수 있는 공간 보다 더 많이 필요하다면, 결과를 보관할 수 있는 더 큰 데이터 유형을 사용할 수도 있습니다. 예를 들어, CHAR(250) 값이 변환된 문자열을 보관하기 위하여 500 바이트를 필요로 한다면, CHAR 값의 최대 길이는 254 바이트이므로 여기에 맞지 않을 것입니다. 그러나 변환 후 결과를 수용하기 위해 VARCHAR(500)을 사용할 수 있습니다. 595 페이지의 『데이터 유형 한계를 초과한 문자 변환』에서 자세한 내용을 참조하십시오.

3 temp_target_length는 결과에 올바른 길이입니다.

데이터베이스 연결시 및 위의 계산에서 리턴된 SQLERRD(1)과 SQLERRD(2) 값을 사용하여, 사용자는 문자열의 길이가 문자 변환 결과 증가할 것인지 감소할 것인지를 판별할 수 있습니다. 일반적으로 0이나 1 값은 확장되지 않음을 나타냅니다. 1보다 큰 값은 길이가 확장될 가능성을 나타내며, 음수 값은 감소할 가능성을 나타냅니다. ‘0’ 값은 다운 레벨 DB2 Universal Database 제품에서만 올 수 있음에 주의하십시오. 또한 이 값들은 다른 데이터베이스 서버 제품에 대하여 정의되지 않습니다. 590 페이지의 표25에서는 DB2 Universal Database를 사용할 때, 다양한 응용프로그램 코드 페이지와 데이터베이스 코드 페이지의 조합에 대한 기대 값을 나열합니다.

표 25. CONNECT에서의 SQLCA.SQLERRD 설정값

응용프로그램 코드 페이지	데이터베이스 코드 페이지	SQLERRD(1)	SQLERRD(2)
SBCS	SBCS	+1	+1
DBCS	DBCS	+1	+1
eucJP	eucJP	+1	+1
eucJP	DBCS	-1	+2
DBCS	eucJP	+2	-1
eucTW	eucTW	+1	+1
eucTW	DBCS	-1	+2
DBCS	eucTW	+2	-1
eucKR	eucKR	+1	+1
eucKR	DBCS	+1	+1
DBCS	eucKR	+1	+1
eucCN	eucCN	+1	+1
eucCN	DBCS	+1	+1
DBCS	eucCN	+1	+1

데이터베이스 서버에서의 확장: SQLERRD(1) 항목이 데이터베이스 서버에서 확장을 나타내는 경우, 사용자의 응용프로그램에서는 클라이언트에서 유효한 길이에 종속적인 문자 세트가 변환 후 데이터베이스 서버에서 유효하지 않을 수 있는 가능성을 고려해야 합니다. 예를 들어, DB2 제품에서는 컬럼 이름 길이가 128 바이트를 초과해서는 안됩니다. DBCS 코드 페이지에서 길이가 128 바이트인 문자열이 EUC 코드 페이지로 변환된 후 128 바이트 한계를 넘어서 확장될 가능성이 있습니다. 이것은 응용프로그램 코드 페이지와 데이터베이스 코드 페이지가 동일할 때는 유효하고, 이들이 다를 때는 유효하지 않은 활동들이 있음을 의미합니다. EUC와 DBCS 데이터베이스를 서로 다른 코드 페이지 상황에서 설계할 때는 주의를 기울여야 합니다.

응용프로그램에서의 확장: SQLERRD(2) 항목이 클라이언트 응용프로그램에서의 확장을 나타내는 경우, 사용자의 응용프로그램에서는 길이에 종속적인 문자 데이터가 변환후 확장될 가능성을 고려해야 합니다. 예를 들어, CHAR(128) 컬럼을 갖는 행이 검색되었습니다. 데이터베이스와 응용프로그램의 코드 페이지가 동일한 상황에서는 리턴된 데이터 길이는 128 바이트입니다. 그러나 동일하지 않은 코드 페

이지 상황에서는 DBCS 코드 페이지에서 코드화된 128 바이트의 데이터는 EUC 코드 페이지로 변환되면 128 바이트를 초과하여 확장될 수 있습니다. 그러므로 완벽한 문자열을 검색하려면 추가적인 저장영역이 할당되어야 합니다.

클라이언트에 기초한 매개변수 검증: 클라이언트와 서버간의 잠재적인 문자 데이터 확장 및 감소의 중요한 부가작용은 클라이언트 응용프로그램과 데이터베이스 서버간에 전달된 데이터의 검증에 관련된 것입니다. 동일하지 않은 코드 페이지 상황에서, 클라이언트에서 유효한 것으로 판명된 데이터가 실제로 문자 변환 후에 데이터베이스 서버에서 유효하지 않을 수도 있습니다. 거꾸로 클라이언트에서 유효하지 않은 문자가 변환 후에 데이터베이스 서버에서 유효할 수도 있습니다.

어떤 일반 사용자 응용프로그램이나 API 라이브러리에서든, 동일하지 않은 코드 페이지 상황에서 일어날 수 있는 문제를 처리할 수 없을 잠재적인 가능성이 있습니다. 또한 문자열 길이와 같은 일부 매개변수 검증이 명령과 API에 대해 클라이언트에서 수행되는 동안, SQL문 내의 토큰이 데이터베이스의 코드 페이지로 변환될 때까지 검증되지 않습니다. 이것은 테이블과 같은 데이터베이스 오브젝트에 액세스하기 위해 동일하지 않은 코드 페이지 환경에서 SQL문을 사용하는 것이 가능한 상황이 될 수도 있지만, 특정 명령이나 API를 사용하여 같은 오브젝트를 액세스하는 것은 불가능합니다.

일반 사용자가 제공하는 테이블에 들어 있는 데이터를 리턴하고, 이 테이블 이름이 128 바이트보다 길지 않은지를 확인하는 응용프로그램을 생각해 봅시다. 이 응용프로그램의 경우 다음과 같은 시나리오가 가능합니다.

1. DBCS 데이터베이스가 작성됩니다. DBCS 클라이언트에서 테이블(t1)이 길이가 128 바이트인 테이블 이름을 가지고 작성됩니다. 이 테이블 이름에는 문자열이 EUC로 변환되는 경우 길이가 2 바이트를 초과하는 일부 문자들을 포함하고 있고, 테이블 이름을 EUC 표기로 하는 경우, 전체 131 바이트 길이가 됩니다. DBCS에서 DBCS로의 연결의 경우 확장되지 않으므로, 데이터베이스 환경에서 테이블 이름은 128 바이트이고 CREATE TABLE은 성공합니다.
2. EUC 클라이언트가 DBCS 데이터베이스에 연결됩니다. EUC로 코드화 될 때는 120 바이트이고 DBCS로 변환될 때는 100 바이트인 테이블 이름을 갖는 테이블(t2)을 작성합니다. DBCS 데이터베이스의 테이블 이름은 100 바이트입니다. CREATE TABLE은 성공적입니다.

3. EUC 클라이언트는 64개의 EUC 문자(131 바이트)로 구성된 테이블 이름을 갖는 테이블(t3)을 작성합니다. 이 이름이 DBCS로 변환되면, 그 길이는 128 바이트 한계로 줄어듭니다. CREATE TABLE은 성공적입니다.
4. EUC 클라이언트가 DBCS 데이터베이스에서 각 테이블(t1, t2 및 t3)에 대하여 응용프로그램을 호출하고, 그 결과는 다음과 같습니다.

테이블	결과
t1	응용프로그램은 테이블 이름이 131 바이트 길이이므로 이것이 유효하지않다고 간주합니다.
t2	올바른 결과를 표시합니다.
t3	응용프로그램은 테이블 이름이 131 바이트 길이이므로 이것이 유효하지않다고 간주합니다.

5. EUC 클라이언트는 CLP에서 DBCS 데이터베이스를 조회하는 데 사용됩니다. 테이블 이름이 클라이언트에서 131 바이트라고 하더라도, 테이블 이름이 서버에서는 128 바이트이므로 조회는 성공합니다.

DESCRIBE문 사용: EUC 데이터베이스에 대하여 수행되는 DESCRIBE는 데이터베이스의 컬럼 정의에 기초하여 혼합 문자 및 GRAPHIC 컬럼에 대한 정보를 리턴합니다. 이 정보는 클라이언트의 코드 페이지로 변환되기 전에 서버의 코드 페이지에 기초합니다.

응용프로그램 문맥에서 해석되는 선택 목록 항목에 대하여 DESCRIBE를 수행할 때(예를 들어 `VALUES SUBSTR(?,1,2)`), 연관된 문자나 그래픽 데이터에 대하여 리턴된 SQLLEN 값을 리턴된 코드 페이지와 함께 평가해야 합니다. 리턴된 코드 페이지가 응용프로그램 코드 페이지와 동일한 경우, 확장되지 않습니다. 리턴된 코드 페이지가 데이터베이스 코드 페이지와 같다면, 확장될 가능성이 있습니다. FOR BIT DATA(코드 페이지 0)이거나 응용프로그램 코드 페이지에 있는 선택 목록 항목은 응용프로그램으로 리턴될 때 변환되지 않으므로, 보고된 길이의 증감이 발생하지 않습니다.

DBCS 데이터베이스를 사용하는 EUC 응용프로그램: 사용자의 응용프로그램 코드 페이지가 EUC코드 페이지이고, DBCS 코드 페이지를 갖는 데이터베이스에 대하여 DESCRIBE를 발행한 경우, CHAR과 GRAPHIC 컬럼에 대하여 리턴된 정보는 데이터베이스 문맥에 리턴됩니다. 예를 들어, DESCRIBE의 일부로 리턴된

CHAR(5) 컬럼에는 SQLLEN 필드에 5 값이 들어 있습니다. 비 EUC 데이터의 경우, 사용자는 이 컬럼에서 데이터를 페치할 때 5 바이트의 저장영역을 할당합니다. EUC 데이터를 사용하는 경우, 해당되지 않습니다. DBCS에서 EUC로의 코드 페이지 변환이 발생할 때, CHAR 컬럼의 문자에 사용된 서로 다른 코드화로 인해 데이터 길이가 증가할 수 있습니다. 예를 들어 대만어 문자 세트의 경우, 최대로 증가하게 되는 길이는 원래 길이의 2배입니다. 즉 DBCS 코드화에서 최대 문자 길이 2 바이트는 EUC에서 최대 4 바이트 길이로 증가할 수 있습니다. 일본어 코드 세트의 경우에도 최대 증가 길이는 2배입니다. 그러나 일본어 DBCS에서의 최대 문자 길이가 2 바이트인 반면, 일본어 EUC에서의 최대 문자 길이는 3 바이트로 증가할 수도 있습니다. 이러한 증가는 인수 1.5에 의해서만 가능할 것으로 보이지만, 일본어 DBCS의 1 바이트 Katakana 문자는 1 바이트 길이이고, 이들이 일본어 EUC에서는 2 바이트입니다. 최대 크기를 판별하는 것에 대한 정보는 576 페이지의『문자 변환 확장 인수』의 내용을 참조하십시오.

문자 변환 결과 발생할 수 있는 데이터 길이의 변경은 혼합 문자 데이터에만 적용됩니다. 그래픽 문자 데이터 코드화는 코드화 체계에 관계없이 항상 2 바이트의 같은 길이입니다. 데이터 유실을 방지하기 위해, 동일하지 않은 코드 페이지 상황이 존재하는지 여부와 이것이 EUC 응용프로그램과 DBCS 데이터베이스 간의 상황인지 여부를 판별해야 합니다. CONNECT문으로부터 리턴되는 SQLCA의 토큰으로부터 데이터베이스 코드 페이지와 응용프로그램 코드 페이지를 판별할 수 있습니다. 더 자세한 정보는 566 페이지의『코드 페이지 값 구하기』또는 *SQL 참조서* 책을 참조하십시오. 그러한 상황이 존재하는 경우, 사용자의 응용프로그램은 해당 코드화 체계의 최대 확장 인수에 근거하여 혼합 문자 데이터를 위한 추가적인 저장영역을 할당해야 합니다.

EUC 데이터베이스를 사용하는 DBCS 응용프로그램: 사용자의 응용프로그램 코드 페이지가 DBCS 코드 페이지이고 EUC 데이터베이스에 대하여 DESCRIBE를 발행한 경우, 592 페이지의『DBCS 데이터베이스를 사용하는 EUC 응용프로그램』에서와 유사한 상황이 발생합니다. 그러나 이 경우, 응용프로그램에서는 SQLLEN 필드의 값에서 나타내는 것보다 더 작은 저장영역을 필요로 합니다. 이 상황에서의 최악의 경우는 EUC에서 모든 데이터가 1 바이트이거나 2 바이트인 것으로, 이것은 DBCS 코드화 체계 하에서 동일한 SQLLEN 바이트를 필요로 함을 의미합

니다. 다른 상황에서는, SQLLEN 바이트보다 더 작은 저장영역이 필요한데, 이는 EUC 문자를 저장하는 데에는 최대 2 바이트가 필요하기 때문입니다.

고정 또는 가변 길이 데이터 유형 사용: DBCS와 EUC 코드 페이지 간에 변환 시 문자열 길이가 변할 수 있으므로, 고정 길이 데이터 유형은 사용하지 않도록 하십시오. 공백 패딩을 필요로 하는지 여부에 따라, DESCRIBE를 수행한 후 SQLTYPE을 고정 길이 문자열에서 가변 길이 문자열로 변경하는 것을 고려해야 합니다. 예를 들어 EUC에서 DBCS로의 연결시 최대 확장 인수가 2인 경우, 응용프로그램은 10 바이트를 할당해야 합니다(592 페이지의 『DBCS 데이터베이스를 사용하는 EUC 응용프로그램』의 CHAR(5) 예에 기초하여).

SQLTYPE이 고정 길이인 경우, EUC 응용프로그램은 코드 페이지 변환 결과 데이터 요소가 최대 크기까지 늘어나지 않은 경우, 공백 패딩을 하여 DBCS 데이터로부터 변환된 EUC 데이터 스트림으로 컬럼을 수신합니다(이것 자체는 5 바이트의 후미 공백 패드를 가질 수 있습니다). SQLTYPE이 가변 길이인 경우, CHAR(5) 컬럼 내용의 원래 의미가 보존되지만, 5 바이트의 소스는 5에서 10 바이트 사이의 목표를 갖게 됩니다. 마찬가지로 데이터 감소의 가능성이 있는 경우에(DBCS 응용프로그램과 EUC 데이터베이스), 가변 길이 데이터 유형으로 작업할 것을 고려해야 합니다.

추가 공간을 할당하거나 데이터 유형을 승격시키는 것을 대체할 수 있는 방법은 데이터를 프래그먼트로 선택하는 것입니다. 예를 들어, 변환 후 길이가 약 6000 바이트가 될 VARCHAR(3000)을 선택하려면, `SUBSTR(VC3000, 1, LENGTH (VC3000)/2)` 및 `SUBSTR(VC3000, (LENGTH(VC3000)/2)+1)`의 두 가지 선택을 수행하여 이들을 2개의 VARCHAR(3000) 응용프로그램 영역에 각각 둡니다. 이 방법은 데이터 유형을 더 이상 승격할 수 없을 때에만 사용할 수 있는 방법입니다. 예를 들어, 최대 2 기가바이트의 길이를 갖는 일본어 DBCS 코드 페이지에서 코드화된 CLOB는 일본어 EUC 코드 페이지로 코드화되면 최대 두 배까지 확장될 가능성이 있습니다. 이것은 이 데이터가 2 기가바이트가 넘는 길이를 지원하는 데이터 유형이 없기 때문에 프래그먼트로 나누어야 함을 의미합니다.

문자 변환 문자열 길이 오버플로우: EUC와 DBCS의 동일하지 않은 코드 페이지 환경에서, 변환이 일어난 후, 전체 문자열을 수용할 충분한 공간이 컬럼에 할당되지 않는 상황이 발생할 수 있습니다. 이 경우, 최대 확장은 문자열 길이의 두

배가 됩니다. 확장시 컬럼 용량을 초과하게 되는 경우, SQLCODE -334(SQLSTATE 22524)가 리턴됩니다.

이 상황은 아래와 같이 즉시 명확해지거나 이전에 고려할 수 있는 상황이 아닐 수 있습니다.

- SQL문이 더 이상 32 765 바이트가 아닐 수도 있습니다. 명령문이 복잡하거나 변환시 확장될 가능성이 많은 상수나 데이터베이스 오브젝트 이름을 사용하는 경우, 예상보다 일찍 이 한계에 도달하게 될 수 있습니다.
- 변환시 short 식별자는 8 바이트까지 및 long 식별자는 128 바이트까지로 최대 길이를 확장하도록 SQL 식별자가 허용됩니다.
- 변환시 호스트 언어 식별자가 최대 255 바이트까지로 확장되도록 허용됩니다.
- SQLCA 구조의 문자 필드가 변환될 때, 이들은 이들의 정의된 최대 길이를 초과하여 확장하도록 허용되지 않습니다.

문자열 변환 규칙: 혼합 코드 페이지 환경에서 응용프로그램을 설계하는 경우, 다음과 같은 상황에 대해서는 *SQL 참조서*를 참조하십시오.

- 설정 조작을 하는 완전 선택에서의 해당 문자열 컬럼(UNION, INTERSECT 및 EXCEPT)
- 병합 피연산자
- 술어 피연산자(LIKE의 예외가 있는)
- CASE문의 결과 표현식
- 스칼라 함수 COALESCE(와 VALUE)의 인수
- IN 술어의 IN 목록의 표현식 값
- 복수 행 VALUES절의 해당 표현식

이러한 상황에서 변환은 데이터베이스 코드 페이지 대신 응용프로그램 코드 페이지에서 발생할 수 있습니다.

데이터 유형 한계를 초과한 문자 변환: EUC와 DBCS의 동일하지 않은 코드 페이지 환경에서 변환이 일어난 후, 혼합 문자 또는 그래픽 문자열의 길이가 해당 데이터 유형에 허용되는 최대 길이를 초과하는 상황이 발생할 수 있습니다. 문자열의 길이가 확장 후에 데이터 유형의 한계를 초과하는 경우, 유형 승격이 발생하지 않습니다. 대신 확장 길이에 허용되는 최대값이 초과되었다는 오류 메시지가 리턴됩니다. 이러한 상황은 삽입에서 보다는 술어를 평가하는 동안에 발생할 가능성이

더 많습니다. 삽입의 경우, 컬럼 폭은 응용프로그램에서 미리 알고 있고 최대 확장 인수도 미리 고려할 수 있습니다. 많은 경우 문자 변환의 이러한 부가 작용은 해당 값을 더 긴 최대 길이를 갖는 연관된 데이터 유형으로 변환함으로써 피할 수 있습니다. 예를 들어, CHAR 값의 최대 길이는 254바이트이고, VARCHAR의 최대 길이는 32672바이트입니다. 확장이 데이터 유형의 최대 길이를 초과하는 경우, 오류 SQLCODE -334(SQLSTATE 22524)가 리턴됩니다.

저장 프로시듀어에서의 문자 변환: 응용프로그램과 데이터베이스의 코드 페이지가 다른 경우, `sqleproc()`나 SQL CALL 호출에서 호스트 변수와 SQLDA에 지정된 혼합 문자나 그래픽 데이터가 변환됩니다. 변환 결과 문자열 길이 확장이 일어나는 경우, 이러한 확장을 처리할 공간이 충분하지 않으면, SQLCODE -334(SQLSTATE 22524)를 수신합니다. 그러므로 저장 프로시듀어를 개발할 때는 문자열 확장을 고려하여 충분한 공간을 확보하도록 해야 합니다. 확장을 허용하기 위해 충분한 공간이 허용된 가변 길이 데이터 유형을 사용해야 합니다.

유니코드(UCS-2) 데이터베이스에 연결된 응용프로그램

앞의 586 페이지의『혼합 코드 세트 환경에서의 개발』에서 설명된 정보가 UCS-2 데이터베이스에도 적용됨에 주의하십시오.

어떤 코드 페이지 환경의 응용프로그램이든 유니코드 데이터베이스에 연결될 수 있습니다. 유니코드 데이터베이스에 연결된 응용프로그램의 경우, 데이터베이스 관리 프로그램이 응용프로그램 코드 페이지와 데이터베이스 코드 페이지(UTF-8)간의 문자열 데이터를 변환합니다. UCS-2 데이터베이스의 경우, GRAPHIC 데이터는 **항상** UCS-2입니다. 그러나 명령행 처리기를 사용하여 그래픽 데이터를 검색하는 경우, 그래픽 문자들은 클라이언트 코드 페이지로도 변환됩니다. 이러한 변환으로 명령행 처리기는 현재의 글꼴로 그래픽 문자를 표시할 수 있습니다. 데이터베이스 관리 프로그램이 UCS-2 문자를 클라이언트 코드 페이지로 변환할 때마다 데이터 유실이 일어날 수 있습니다. 데이터베이스 관리 프로그램이 클라이언트 코드 페이지의 유효한 문자로 변환할 수 없는 문자들은 해당 코드 페이지의 기본 대체 문자로 대체됩니다.

DB2가 코드 페이지에서 UTF-8로 문자들을 변환할 때, 문자들을 나타내는 전체 바이트 수가 그 문자들의 코드 페이지와 코드 포인트에 따라 늘어나거나 줄어들 수 있습니다. 7 비트 ASCII는 UTF-8에서 변하지 않고 그대로 있으며, 각 ASCII

문자들은 1 바이트를 필요로 합니다. 비 ASCII UCS-2 문자들은 각각 2 바이트나 3 바이트가 됩니다. UTF-8 변환에 대한 정보는 *관리 안내서* 또는 유니코드 표준 문서를 참조하십시오.

유니코드 데이터베이스로 연결된 응용프로그램의 경우, GRAPHIC 데이터는 이미 유니코드에 있습니다. DBCS 데이터베이스에 연결된 응용프로그램의 경우, GRAPHIC 데이터가 응용프로그램 DBCS 코드 페이지와 데이터베이스 DBCS 코드 페이지간에 변환됩니다. 유니코드 응용프로그램은 그래픽 데이터를 위해 유니코드 자신들 사이에서 필요한 변환을 수행하거나 WCHARTYPE CONVERT 옵션을 설정하고 `wchar_t`를 사용해야 합니다. 이 옵션에 대한 세부사항은 694 페이지의 『C 및 C++에서의 그래픽 호스트 변수 처리』의 내용을 참조하십시오.

다중 사이트 갱신시 고려사항

이 절에서는 사용자 응용프로그램이 원격 데이터베이스에 대하여 작업하는 방법과 한 번에 둘 이상의 데이터베이스에 대하여 작업하는 방법을 설명합니다. 다음과 같은 내용이 포함됩니다.

- 원격 작업 단위(RUOW)
- 다중 사이트 갱신

DB2를 사용하여, 사용자는 BACKUP, RESTORE, DROP DATABASE, CREATE DATABASE와 같은 원격 서버 함수들을 지역 응용프로그램처럼 수행시킬 수 있습니다. 이들 함수를 원격으로 사용하는 것에 대한 정보는 *관리 안내서* 책을 참조하십시오.

원격 작업 단위(RUOW)

작업 단위(UOW)는 하나의 논리적 트랜잭션입니다. 이것은 일련의 SQL문들로 구성되어, 이러한 조작이 모두 성공적으로 수행되거나, 전체가 모두 성공하지 못한 것으로 처리됩니다.

원격 작업 단위(RUOW)를 통해 사용자나 응용프로그램은 작업 단위(UOW)별로 한 위치에서 데이터를 읽거나 갱신할 수 있습니다. 이것은 작업 단위(UOW) 내에

서의 하나의 데이터베이스에 대한 액세스를 지원합니다. 응용프로그램이 여러개의 원격 데이터베이스에 액세스할 수 있는 반면, 작업 단위(UOW) 내에서는 하나의 데이터베이스에만 액세스할 수 있습니다.

원격 작업 단위(RUOW)는 다음과 같은 특성을 갖습니다.
- 작업 단위(UOW)별 복수 요청이 지원됩니다.
- 작업 단위(UOW)별 복수 커서가 지원됩니다.
- 각 작업 단위(UOW)는 하나의 데이터베이스에만 액세스할 수 있습니다.
- 응용프로그램은 작업 단위(UOW)를 확약하거나 구간 복원할 수 있습니다. 특정 오류 상황에서 서버는 작업 단위(UOW)를 구간 복원합니다.

다중 사이트 갱신

분산 작업 단위(DUOW) 및 2단계 확약이라고도 하는 다중 사이트 갱신은 응용프로그램이 무결성을 보장받으면서 복수의 원격 데이터베이스 서버에 있는 데이터를 갱신할 수 있도록 하는 기능입니다. 다중 사이트 갱신의 좋은 예는 다른 데이터베이스 서버에서 한 계좌에서 다른 계좌로 돈을 이체시키는 것과 같은 은행 트랜잭션입니다. 이러한 트랜잭션에서, 하나의 계좌에서 출금 조작을 구현하는 갱신이 다른 계좌로의 입금 처리를 필요로 하는 갱신 역시 확약되지 않는 한, 확약되지 않습니다. 다중 사이트 갱신의 고려사항은 이러한 계좌를 나타내는 데이터가 두 개의 서로 다른 데이터베이스 서버에 의해 관리될 때 적용됩니다.

작업 단위(UOW) 내에서 복수의 DB2 Universal Database 데이터베이스를 읽고 갱신하는 데 다중 사이트 갱신을 사용할 수 있습니다. DB2 Connect를 설치했거나 DB2 Universal Database Enterprise Edition에서 제공하는 DB2 Connect 기능을 사용하면 호스트와의 다중 사이트 갱신이나 OS/390용 DB2 Universal Database 및 AS/400용 DB2 Universal Database과 같은 AS/400 데이터베이스 서버를 사용할 수도 있습니다. 다른 데이터베이스 서버와 다중 사이트 갱신을 사용할 때는 897 페이지의 『DB2 Connect와 다중 사이트 갱신』에서 설명된 바와 같이 특정 제한사항이 적용됩니다.

트랜잭션 관리 프로그램은 복수의 데이터베이스간의 확약을 조정합니다. CICS가 트랜잭션 처리(TP) 모니터 환경을 사용하는 경우, TP 모니터는 자신의 트랜잭션 관리 프로그램을 사용합니다. 그렇지 않은 경우 DB2에서 제공하는 트랜잭션 관리 프로그램이 사용됩니다. OS/2용 DB2 Universal Database는 XA(확장 아키텍처)를

따르는 자원 관리 프로그램입니다. DB2 Connect를 통해 액세스하는 호스트 및 AS/400 데이터베이스 서버는 XA 호환 자원 관리자입니다. 또한 DB2 Universal Database 트랜잭션 관리 프로그램이 XA를 따르는 트랜잭션 관리 프로그램이 *아니므로* 트랜잭션 관리 프로그램만이 DB2 데이터베이스를 조정할 수 있습니다.

다중 사이트 갱신에 대한 세부사항은 *관리 안내서* 책을 참조하십시오.

다중 사이트 갱신 사용 시점

다중 사이트 갱신은 둘 이상의 데이터베이스에 대하여 작업하면서 데이터 무결성을 유지보수하고자 할 때 가장 유용합니다. 예를 들어 은행 각 지점이 각각의 데이터베이스를 가지고 있는 경우, 계좌 이체 응용프로그램은 다음을 수행할 수 있습니다.

- 송신자의 데이터베이스에 연결합니다.
- 송신자의 계좌 잔금을 읽고 잔액이 충분한지를 확인합니다.
- 이체할 금액을 송신자 계좌 잔액에서 뺍니다.
- 수신자의 데이터베이스에 연결합니다.
- 수신자의 계좌 잔액에 이체 금액을 더합니다.
- 데이터베이스를 확약합니다.

하나의 작업 단위(UOW) 안에서 수행함으로써, 두 데이터베이스가 모두 갱신되거나 모두 갱신되지 않습니다.

다중 사이트 갱신 응용프로그램을 위한 SQL 코딩

600 페이지의 표26에서는 다중 사이트 갱신을 위한 SQL문 코딩 방법이 나와 있습니다. 왼쪽의 컬럼에서는 다중 사이트 갱신을 사용하지 않는 SQL문을 보여주고, 오른쪽 컬럼에서는 다중 사이트 갱신을 사용하는 유사한 명령문을 보여줍니다.

표 26. RUOW 및 다중 사이트 갱신 SQL문

RUOW문	다중 사이트 갱신 명령문

CONNECT TO D1
  SELECT
  UPDATE
   COMMIT
 	
CONNECT TO D2
  INSERT
   COMMIT
 	
CONNECT TO D1
  SELECT
   COMMIT
CONNECT RESET
	CONNECT TO D1
	  SELECT
	  UPDATE
	
	CONNECT TO D2
	  INSERT
	  RELEASE CURRENT
	
	SET CONNECTION D1
	  SELECT
	  RELEASE D1
	   COMMIT

왼쪽 컬럼의 SQL문은 각 작업 단위(UOW)에 대하여 하나의 데이터베이스에만 액세스합니다. 이것은 원격 작업 단위(RUOW) 응용프로그램입니다.

오른쪽 컬럼의 SQL문은 작업 단위(UOW) 내에서 둘 이상의 데이터베이스에 액세스합니다. 이것은 다중 사이트 갱신 응용프로그램입니다.

일부 SQL문은 다중 사이트 갱신 응용프로그램에서 달리 코딩되고 해석됩니다.

- 현재의 작업 단위(UOW)는 다른 데이터베이스에 연결하기 전에 확약되거나 구간 복원될 필요가 없습니다.
- 다른 데이터베이스에 연결될 때 현재의 연결이 해제되지 않습니다. 대신 이것은 *휴면(dormant)* 상태에 들어갑니다. CONNECT문이 실패해도 현재 연결에 영향을 미치지 않습니다.
- 데이터베이스에 대한 현재 연결 또는 휴면 연결이 존재하는 경우, USER/USING 절을 사용하여 연결할 수 없습니다.
- SET CONNECTION문을 사용하여 휴면 연결을 현재 연결로 변경할 수 있습니다.

  또한 휴면 데이터베이스로 CONNECT문을 발행하여 마찬가지로 수행할 수 있습니다. SQLRULES를 STD로 설정하는 경우 허용되지 않습니다. 사전 처리 컴

파일러 옵션이나 SET CLIENT 명령이나 API를 사용하여 SQLRULES 값을 설정할 수 있습니다. SQLRULES (DB2)의 기본값으로 CONNECT문을 사용하여 연결을 전환할 수 있습니다.

- 선택에서, 다른 데이터베이스로 전환했다가 다시 원래의 데이터베이스로 돌아오는 경우 커서 위치는 영향을 받지 않습니다.
- CONNECT RESET문은 현재의 연결을 해제하지 않고 내재적으로 현재 작업 단위를 확약하지 않습니다. 대신, 이것은 기본 데이터베이스가 정의되어 있는 경우, 여기에 명시적으로 연결하는 것과 같습니다. 내재된 연결이 정의되어 있지 않은 경우, SQLCODE -1024(SQLSTATE 08003)가 리턴됩니다.
- 다음 COMMIT에서 연결 해제에 대한 연결을 표시하려면 RELEASE문을 사용할 수 있습니다. RELEASE CURRENT문은 현재의 연결에 적용되고, RELEASE 연결은 명명된 연결에 적용되며, RELEASE ALL문은 모든 연결에 적용됩니다.

  릴리스로 표시된 연결은 다음 COMMIT문에서 제거될 때까지 여전히 사용될 수 있습니다. 구간 복원은 연결을 제거하지 않습니다. 이것을 통해 연결 재시도를 수행할 수 있습니다. DISCONNECT문(또는 사전 처리 컴파일러 옵션)을 사용하여 확약이나 구간 복원 후에 연결을 제거하십시오.
- COMMIT문은 현재 또는 휴면 작업 단위에서의 모든 데이터베이스를 확약합니다.
- ROLLBACK문은 작업 단위 내 모든 데이터베이스를 구간 복원하고, 작업 단위 내에서 액세스되었는지 여부에 상관없이 모든 데이터베이스에 대한 보류된 커서를 닫습니다.
- (휴면 연결 및 릴리스로 표시된 연결을 포함한)모든 연결은 응용프로그램 프로세스가 종료하면 연결해제됩니다.
- (현재의 연결을 조회하기만 하는 옵션 없는 CONNECT문을 포함하여) 성공적인 연결 횟수가 SQLCA의 SQLERRD(3)과 SQLERRD(4) 필드에 리턴됩니다.

  SQLERRD(3) 필드는 연결된 데이터베이스가 현재 작업 단위(UOW)에서 갱신 가능한지 여부에 대한 정보를 리턴합니다. 가능한 값은 다음과 같습니다.

  1 갱신 가능.

  2 읽기 전용.

SQLERRD(4) 필드는 현재의 연결 특성에 대한 다음과 같은 정보를 리턴합니다.

0 적용 불가능. 이 상태는 1 단계 확약을 사용하고 갱신자인 하위 레벨 클라이언트로부터 수행되는 경우에만 가능합니다.

1 1 단계 확약

2 1 단계 확약(읽기 전용). 이 상태는 DB2 Connect 동기점 관리자를 시작하지 않고 DB2 Connect를 사용하여 액세스할 수 있는 호스트나 AS/400 데이터베이스 서버에만 적용될 수 있습니다.

3 2 단계 확약.

도구나 유틸리티를 작성하고 있는 경우, 연결이 읽기 전용일 때 사용자들에게 메시지를 발행하고자 할 수 있습니다.

다중 사이트 갱신 응용프로그램 사전 처리 컴파일

다중 사이트 갱신 응용프로그램을 사전 처리 컴파일하는 경우, 반드시 CLP 연결을 유형 1 연결로 설정해야 합니다. 그렇지 않은 경우, 응용프로그램을 사전 처리 컴파일하려고 할 때 SQLCODE 30090(SQLSTATE 25000)을 받게 됩니다. 연결 유형 설정 정보에 대해서는 *Command Reference* 책을 참조하십시오. 다음 사전 처리 컴파일러 옵션은 다중 사이트 갱신을 사용하는 응용프로그램을 사전 처리 컴파일할 때 사용됩니다.

CONNECT (1 | 2)

599 페이지의『다중 사이트 갱신 응용프로그램을 위한 SQL 코딩』에서 설명된 것처럼, CONNECT 2를 지정하여, 이 응용프로그램이 다중 사이트 갱신 응용프로그램용 SQL 구문을 사용함을 나타내십시오. 기본값인 CONNECT 1은 응용프로그램에 적용되는 SQL 구문에 대한 일반적인 (RUOW) 규칙들입니다.

SYNCPOINT (ONEPHASE | TWOPHASE | NONE)

SYNCPOINT TWOPHASE를 지정하고 DB2가 트랜잭션을 조정하면 DB2는 트랜잭션 상태 정보를 유지보수해야 합니다. 응용프로그램을 전개할 때 데이터베이스 관리자 구성 매개변수인 TM_DATABASE를 구성하여 이 데이터베이스를 정의해야 합니다. TM_DATABASE에 대한 정보는 *관리 안*

*내서*에서 자세한 내용을 참조하십시오. 이들 SYNCPOINT 옵션이 프로그램 작동 방식에 영향을 미치는 방법에 관한 정보는 *SQL 참조서*의 개념 절을 참조하십시오.

SQLRULES (DB2 | STD)
: 다중 사이트 갱신 응용프로그램에서 DB2 규칙을 사용할 것인지 아니면, ISO/ANSI SQL92에 기초한 표준(STD) 규칙을 사용할 것인지 여부를 지정합니다. DB2 규칙을 통해 사용자는 CONNECT문을 휴면 데이터베이스에 발행할 수 있습니다. STD 규칙에서는 불가능합니다.

DISCONNECT (EXPLICIT | CONDITIONAL | AUTOMATIC)
: COMMIT을 수행할 때 어떤 데이터베이스 연결이 해제될 것인지를 지정합니다. 즉, RELEASE문을 사용하여 릴리스로 표시된 데이터베이스인지(EXPLICIT), 열려 있는 WITH HOLD 커서가 없는 모든 데이터베이스인지(CONDITIONAL) 또는 모든 연결인지(AUTOMATIC)를 지정합니다.

이 사전 처리 컴파일러의 자세한 설명은 *Command Reference* 책을 참조하십시오.

다중 사이트 갱신 사전 처리 컴파일러 옵션은 첫번째 데이터베이스 연결이 이루어졌을 때 효과적입니다. 아무 연결도 설정되기 전이나 모든 연결이 해제된 후, 기존 연결이 없을 때 연결 설정을 대체하기 위해 SET CLIENT API를 사용할 수 있습니다. QUERY CLIENT API를 사용하여 응용프로그램 프로세스의 현재 연결 설정값을 조회할 수 있습니다.

사용자의 응용프로그램에서 참조된 오브젝트가 없는 경우, 바인더가 실패합니다. 다중 사이트 갱신 응용프로그램을 처리하는 방법에는 세 가지가 있습니다.

- 응용프로그램을 여러 파일로 나누어, 각각이 하나의 데이터베이스만을 액세스하도록 합니다. 그리고 나서 각 파일들이 액세스하는 하나의 데이터베이스에 대하여 각 파일들을 준비하고 바인드합니다.
- 각 테이블은 각 데이터베이스에 존재하는지 확인할 수 있습니다. 예를 들어 은행 지점은 각각의 테이블이 동일한(데이터는 제외) 데이터베이스를 가지고 있을 수 있습니다.
- 동적 SQL만을 사용할 수 있습니다.

다중 사이트 갱신 응용프로그램의 구성 매개변수 지정

호스트 또는 AS/400 데이터베이스에 연결된 XA 트랜잭션 관리자로 조정된 다중 사이트 갱신을 수행하는 것에 대한 자세한 내용은 *DB2 Connect 사용자 안내서*을 참조하십시오.

다음 구성 매개변수는 다중 사이트 갱신을 수행하는 응용프로그램에 영향을 줍니다. LOCKTIMEOUT 예외와 함께 구성 매개변수는 데이터베이스 관리자 구성 매개변수입니다. LOCKTIMEOUT은 데이터베이스 구성 매개변수입니다.

TM_DATABASE
어떤 데이터베이스가 2단계 확약 트랜잭션의 트랜잭션 관리 프로그램으로 작동할 것인지를 지정합니다.

RESYNC_INTERVAL
2단계 확약중 이상 실패 트랜잭션을 재동기화를 다시 시도하기 전에 시스템이 기다리는 초 수를 지정합니다. 2단계 확약중 이상 실패 트랜잭션은 2단계 확약 중 첫번째 단계는 성공적으로 완료하였으나 두번재 단계에서 실패한 트랜잭션입니다.

LOCKTIMEOUT
잠금 대기가 시간을 종료하고 현재의 트랜잭션을 구간 복원하기 전까지의 초 수를 지정합니다. 응용프로그램은 명시적 ROLLBACK을 실행하여 다중 사이트 갱신에 참여하는 모든 데이터베이스를 구간 복원해야 합니다. LOCKTIMEOUT은 데이터베이스 구성 매개변수입니다.

TP_MON_NAME
TP 모니터가 있는 경우 그 이름을 지정합니다.

SPM_RESYNC_AGENT_LIMIT
SNA를 사용하여 호스트나 AS/400 서버와 재동기를 수행할 수 있는 동시 에이전트의 수를 지정합니다.

SPM_NAME

- SPM이 TCP/IP 2PC 연결을 가지고 사용된 경우, SPM_NAME은 네트워크 내에서 고유한 식별자여야 합니다. DB2 인스턴스를 작성할 때 DB2는 SPM_NAME의 기본값을 TCP/IP 호스트 이름에서 가져옵니다. 환경에서 이 값을 승인할 수 없을 때는 이 값을 수정할 수 있습니다.

호스트 데이터베이스 서버와의 TCP/IP 연결성에 대해 기본값을 승인해야 합니다. 호스트 또는 AS/400 데이터베이스 서버에 대한 SNA 연결에 대해 이 값은 SNA 제품에서 정의된 SNA LU 프로파일과 일치해야 합니다.

- SPM이 SNA 2PC 연결과 함께 사용되는 경우, SPM 이름은 2PC에 사용된 LU_NAME으로 설정되어야 합니다.
- SPM이 TCP/IP와 SNA에 모두 사용된 경우, 2PC에 사용된 LU_NAME이 사용되어야 합니다.

주: 환경에서 이루어지는 호스트 또는 AS/400 데이터베이스와의 다중 사이트 갱신에서는 SPM이 필요합니다. 자세한 정보는 *DB2 Connect 사용자 안내서* 책을 참조하십시오.

SPM_LOG_SIZE

SPM이 연결, 현재 연결 상태 등에 대한 정보를 기록하기 위해 사용하는 각각의 1차 및 2차 로그 파일의 4 킬로바이트 페이지의 수.

이 구성매개변수에 대한 자세한 설명은 *관리 안내서* 책을 참조하십시오.

다중사이트 갱신 제한사항

다음의 제한사항은 DB2의 다중 사이트 갱신에 적용됩니다.

- TxSeries과 같은 트랜잭션 처리(TP) 모니터 환경에서, DISCONNECT문은 지원되지 않습니다. TP 모니터와 함께 DISCONNECT를 사용하는 경우, SQLCODE -30090(SQLSTATE 25000)을 수신하게 됩니다. DISCONNECT 대신 RELEASE 다음에 COMMIT을 사용하십시오.
- 동적 COMMIT과 ROLLBACK은 연결 유형 2 환경에서 지원되지 않습니다. 이 환경에서 COMMIT을 사용하는 경우 거부되며 SQLCODE -925(SQLSTATE 2D521)가 수신됩니다. 이 환경에서 ROLLBACK을 사용하는 경우 거부되며 SQLCODE -926(SQLSTATE 2D521)이 수신됩니다.
- 사전 처리 컴파일러 옵션 DISCONNECT CONDITIONAL을 버전 1 데이터베이스에 대한 연결에는 사용할 수 없습니다. 버전 1 데이터베이스에 대한 연결은 보류된 커서가 열려있다고 하더라도 COMMIT시에 연결해제됩니다.

- WITH HOLD로 선언된 커서들이 다중 사이트 갱신에서 지원된다고 하더라도, DISCONNECT가 성공하려면, WITH HOLD로 선언된 모든 커서들을 닫고, DISCONNECt 요청 전에 COMMIT을 발행해야 합니다.
- TP 모니터 환경의 서비스가 트랜잭션 관리에 사용될 때, 다중 사이트 갱신 옵션은 내재적으로 CONNECT 유형 2, SYNCPOINT TWOPHASE, SQLRULES DB2, DISCONNECT EXPLICIT입니다. 사전 처리 컴파일이나 SET CLIENT API를 사용하여 이들 옵션을 변경하는 것은 불필요하며, 무시됩니다.
- 다음의 API는 다중 사이트 갱신에서 지원되지 않으므로, 응용프로그램에서 이들 API를 다중 사이트 갱신(CONNECT 유형 2)에 사용하는 경우, SQLCODE -30090(SQLSTATE 25000)을 수신합니다.

```
BACKUP DATABASE
BIND
EXPORT
IMPORT
LOAD
MIGRATE DATABASE
PRECOMPILE PROGRAM
RESTART DATABASE
RESTORE DATABASE
REORGANIZE TABLE
ROLLFORWARD DATABASE
```

- 저장 프로시듀어는 다중 사이트 갱신에서 지원됩니다. 그러나 다중 사이트 갱신(CONNECT 유형 2)에서 COMMIT이나 ROLLBACK문을 발행하는 저장 프로시듀어의 경우, 이 명령문들이 다중 사이트 갱신에서 지원되지 않으므로 SQLCODE -30090(SQLSTATE 25000)을 수신합니다.

Host 또는 AS/400 서버에 액세스

서로 다른 데이터베이스 시스템에 액세스하거나 이를 갱신하는 응용프로그램을 개발하려면, 다음을 수행해야 합니다.

1. 응용프로그램에서 액세스하려는 모든 데이터베이스 시스템에서 지원되는 사전 처리 컴파일/바인드 옵션과 SQL문을 사용하십시오. 예를 들어 저장 프로시듀어는 모든 플랫폼에서 지원되는 것은 아닙니다.

   IBM 제품의 경우, 코딩을 시작하기 **전에** *SQL 참조서* 책을 참조하십시오.

2. 가능하다면, 응용프로그램이 SQLCODE보다는 SQLSTATE를 확인하도록 하십시오.

   응용프로그램에서 DB2 Connect를 사용하고 사용자는 SQLCODE를 사용하려는 경우, DB2 Connect에서 제공하는 맵핑 기능을 사용하여 서로 다른 데이터베이스 사이의 SQLCODE 변환을 고려해 보십시오.

3. 지원하려는 호스트나 AS/400 데이터베이스(예를 들면 OS/390용 DB2 Universal Database, OS/400 또는 VSE & VM용 DB2)에서 응용프로그램을 테스트하십시오. *DB2 Connect 사용자 안내서*에서 자세한 정보를 참조하십시오.

호스트나 AS/400 데이터베이스 시스템 액세스에 대한 정보는 881 페이지의 『부록D. 호스트 또는 AS/400 환경에서 프로그래밍』의 내용을 참조하십시오.

다중 스레드 데이터베이스 액세스

일부 운영 체제에는 하나의 프로세스 내에서 여러개의 실행 스레드를 수행할 수 있는 기능이 있습니다. 이것을 통해 응용프로그램은 비동기 이벤트를 처리할 수 있고, 폴링 스킴으로 재정렬하지 않고도 이벤트 중심의 응용프로그램을 더욱 쉽게 작성할 수 있습니다. 이 절에서는 데이터베이스 관리 프로그램이 여러 스레드에 대하여 작업하는 방식을 논의하고, 염두에 두어야 할 설계시 지침을 나열합니다. 사용자의 플랫폼이 다중 스레드 기능을 지원하는지를 판별하려면, *응용프로그램 빌드 안내서* 책을 참조하십시오.

이 절에서는 사용자가 중요한 절 및 세마포어와 같은 다중 스레드 응용프로그램 개발에 관련된 용어에 익숙하다고 가정합니다. 이들 용어에 익숙하지 않다면 해당 운영 체제의 프로그램 관련 문서를 참조하십시오.

DB2 응용프로그램은 *문맥*을 사용하여 복수 스레드로부터 SQL문을 실행할 수 있습니다. 문맥이란 응용프로그램이 모든 SQL문과 API 호출을 수행하는 환경입니다. 모든 연결, 작업 단위(UOW) 및 기타 데이터베이스 자원들도 모두 특정 문맥과 연관되어 있습니다. 각 문맥은 응용프로그램 내에서 하나 이상의 스레드와 연관되어 있습니다.

문맥 내 실행가능한 각각의 SQL문에서 첫번째 런타임 서비스 호출은 항상 래치를 확보하려고 합니다. 성공하면 처리가 계속됩니다. 같은 문맥의 다른 스레드에 있

는 SQL문이 이미 해당 래치를 가지고 있기 때문에 실패한 경우, 세마포어가 포스트될 때까지 신호를 보내는 세마포어에서 호출은 정지하고, 세마포어가 포스트되는 시점에 호출은 래치를 확보하고 처리를 계속합니다. 이 래치는 SQL문이 처리를 완료할 때까지 확보하고 있으며, 처리가 완료되면, 해당 특정 SQL문에 대하여 생성되는 최종 런타임 서비스 호출에 의해 릴리스됩니다.

다른 스레드들도 동시에 SQL문을 실행시키려고 하지만, 네트 결과는 문맥 내의 각 SQL문이 가장 작은 단위로 실행된다는 것입니다. 이러한 조치는 동시에 서로 다른 스레드로 인해 내부 데이터 구조가 변경되지 않도록 합니다. API들도 역시 런타임 서비스에 의해 사용되는 래치를 사용합니다. 그러므로 API도 각 문맥 내에서 런타임 서비스 루틴과 동일한 제한사항을 갖습니다.

기본적으로 모든 응용프로그램은 모든 데이터베이스 액세스에 사용되는 하나의 문맥을 갖습니다. 이것이 단일 스레드 응용프로그램에 대하여 완벽한 반면, SQL문의 직렬화는 단일 문맥이 다중 스레드 응용프로그램에 부적절하게 만듭니다. 다음과 같은 DB2 API를 사용하여 사용자의 응용프로그램은 별도의 문맥을 각 스레드에 접속하고, 스레드간에 문맥이 전달될 수 있도록 합니다.

- sqleSetTypeCtx()
- sqleBeginCtx()
- sqleEndCtx()
- sqleAttachToCtx()
- sqleDetachFromCtx()
- sqleGetCurrentCtx()
- sqleInterruptCtx()

문맥은 프로세스 내 스레드 간에 교환될 수 있으나 프로세스간에는 교환될 수 없습니다. 여러 문맥 중 하나를 사용함으로써 동시적인 트랜잭션에 대한 지원을 제공합니다. 이들 문맥 API의 사용 방법에 대한 자세한 내용은 *Administrative API Reference* 및 612 페이지의 『동시 트랜잭션』의 내용을 참조하십시오.

다중 스레드 사용시 권장사항

다중 스레드 응용프로그램에서 데이터베이스에 액세스할 때, 다음의 지침을 따르십시오.

- **데이터 구조 변경을 직렬화하십시오.**

응용프로그램에서는 SQL문이나 데이터베이스 관리 프로그램 루틴이 다른 스레드에서 처리되고 있는 동안, 이 SQL문과 데이터베이스 관리 프로그램 루틴이 사용하는 사용자 정의 데이터 구조가 하나의 스레드에 의해 변경되지 않도록 해야 합니다. 예를 들어, SQLDA가 어떤 스레드의 SQL문에 의해 사용되고 있는 동안, 다른 스레드가 이것을 재할당하도록 해서는 안됩니다.

- **별도의 데이터 구조 사용을 고려하십시오.**

  각 스레드에게 각자의 사용자 정의 데이터 구조를 주어 사용을 직렬화하지 않도록 하는 것이 더 쉬울 것입니다. 이것은 특히 모든 실행가능 SQL문에 의해 사용될 뿐 아니라 모든 데이터베이스 관리 프로그램 루틴에 의해 사용되는 SQLCA의 경우 적용됩니다. SQLCA에서의 이러한 문제점을 방지하기 위한 방법으로 세 가지가 있습니다.

  1. EXEC SQL INCLUDE SQLCA를 사용하되, 첫번째 스레드 이외의 다른 모든 스레드가 사용하는 모든 루틴의 첫머리에 `struct sqlca sqlca`를 추가하십시오.
  2. 전역 범위 대신 SQL이 들어 있는 각 루틴 내부에 EXEC SQL INCLUDE SQLCA를 지정하십시오.
  3. EXEC SQL INCLUDE SQLCA를 `#include "sqlca.h"`로 대체하고 SQL을 사용하는 모든 루틴의 첫머리에 `"struct sqlca sqlca"`를 추가하십시오.

코드 페이지 및 국가 코드에 대하여 작업하는 다중 스레드 UNIX 응용프로그램

AIX, Solaris, HP-UX 및 Silicon Graphics IRIX에서 데이터베이스 연결에 사용되는 코드 페이지와 국가 코드의 런타임 조회에사용되는 함수가 변경되었습니다. 이 함수들은 이제 스레드에 안전하지만 많은 수의 동시적인 데이터베이스 연결을 사용하는 다중 스레드 응용프로그램에서는 잠금 경합 및 결과적으로 성능 저하가 나타나게 되었습니다.

다중 스레드 응용프로그램에서 잠금 경합의 가능성을 줄이기 위해 새로운 환경 변수(DB2_FORCE_NLS_CACHE)가 작성되었습니다. DB2_FORCE_NLS_CACHE를 TRUE로 설정하면, 코드 페이지와 국가 코드 정보가 스레드가 맨 처음 액세스할 때 저장됩니다. 이 시점부터 캐쉬된 정보는 이 정

보를 요구하는 다른 스레드에 사용됩니다. 이 정보를 저장함으로써, 잠금 경합은 감소하고, 특정한 경우에 성능상이 향상되기도 합니다.

응용프로그램이 연결간의 로케일 설정값을 변경하는 경우, DB2_FORCE_NLS_CACHE는 TRUE로 설정해서는 안됩니다. 설정하는 경우, 원래의 로케일 정보가 로케일 설정값이 변경된 후에도 리턴됩니다. 일반적으로 다중 스레드 응용프로그램은 로케일 설정을 변경하지 않습니다. 이러한 점으로 응용프로그램이 스레드에 안전하게 있을 수 있습니다.

다중 스레드 사용시 잠재적 함정

다중 스레드를 사용하는 응용프로그램은 당연히 단일 스레드 응용프로그램보다 더 복잡합니다. 이것이 잠재적으로 예상치 않은 문제점을 발생시키기도 합니다. 다중 스레드 응용프로그램을 작성할 때, 다음과 같은 주의를 기울여야 합니다.

- **둘 이상의 문맥 간에 데이터베이스 종속성**

  응용프로그램에서 각 문맥은 데이터베이스 오브젝트에 대한 잠금과 같은 각자의 데이터베이스 자원 세트를 가지고 있습니다. 이러한 점으로 인해 두 개의 문맥이 하나의 데이터베이스 오브젝트에 액세스하는 경우 교착 상태에 놓이게 될 수 있습니다. 데이터베이스 관리 프로그램은 교착 상태를 발견하고, 문맥 중 하나가 SQLCODE-911을 수신하며, 그 작업 단위(UOW)는 구간 복원됩니다.

- **둘 이상의 문맥 간의 응용프로그램 종속성**

  문맥 간에 종속 관계를 설정하는 프로그래밍 기법을 사용하는 경우, 주의해 주십시오. 래치, 세마포어 및 중요한 절 등이 그러한 종속성을 설정하는 프로그래밍 기법의 예입니다. 문맥들 사이에 응용프로그램과 데이터베이스 모두의 종속성을 갖는 두 개의 문맥이 응용프로그램에 있는 경우, 응용프로그램이 교착 상태에 놓일 가능성이 있습니다. 일부 종속성이 데이터베이스 관리 프로그램 외부에 있는 경우, 교착 상태는 발견되지 않으므로 응용프로그램은 일시 중단되거나 정지합니다.

  이러한 종류의 문제점의 예로, 둘다 공통 데이터 구조에 액세스하는 두 개의 문맥이 있는 응용프로그램을 생각해 봅시다. 두 가지 문맥이 동시에 데이터 구조를 변경할 때의 문제점을 피하기 위해, 데이터 구조는 세마포어로 보호됩니다. 문맥은 다음과 같습니다.

```
context 1
SELECT * FROM TAB1 FOR UPDATE....
UPDATE TAB1 SET....
get semaphore
access data structure
release semaphore
COMMIT

context 2
get semaphore
access data structure
SELECT * FROM TAB1...
release semaphore
COMMIT
```

두번째 문맥이 세마포어를 받아 데이터 구조에 액세스하는 동안, 첫번째 문맥이 성공적으로 SELECT와 UPDATE문을 실행했다고 가정하십시오. 이제 첫번째 문맥은 세마포어를 얻으려고 하지만, 두번째 문맥이 세마포어를 가지고 있으므로 얻을 수가 없습니다. 이제 두번째 문맥은 테이블 TAB1으로부터 행을 읽으려고 하지만, 첫번째 문맥이 데이터베이스 잠금을 가지고 있으므로 멈추게 됩니다. 이제 응용프로그램은 Context 2가 완료되기 전에 Context 1이 종료할 수 없고, Context 2는 Context 1이 종료하기를 기다리는 상태가 되었습니다. 응용프로그램이 교착 상태에 빠졌으나 데이터베이스 관리 프로그램이 세마포어 종속성을 모르고 있으므로, 문맥은 구간 복원되지 않습니다. 그래서 응용프로그램이 일시 중단됩니다.

복수 문맥에서 교착 상태 방지

데이터베이스 관리 프로그램이 스레드간의 교착 상태를 발견하지 못하므로, 교착 상태를 방지하거나 최소한 피할 수 있는 방식으로 응용프로그램을 설계하고 코딩하십시오. 위의 예에서, 여러 가지 방법으로 교착 상태를 피할 수 있습니다.

- 세마포어를 얻기 전에 보류되어 있는 잠금을 모두 릴리스하십시오.

  Context 1의 코드를 변경하여 세마포어를 갖기 전에 확약을 수행하십시오.

- 세마포어로 보호되는 절 내부에 SQL문을 코딩하지 마십시오.

  Context 2의 코드를 변경하여 SELECT를 수행하기 전에 세마포어를 릴리스하십시오.

- 세마포어 내에 모든 SQL문을 코딩하십시오.

Context 1의 코드를 변경하여 SELECT문을 수행하기 전에 세마포어를 확보하십시오. 이 기법들이 통하기는 하지만 매우 권장되는 방법은 아닌데, 이는 세마포어가 데이터베이스 관리 프로그램에 대한 액세스를 직렬화하여, 다중 스레드 사용의 이점을 잠재적으로 손상시키기 때문입니다.

- LOCKTIMEOUT 데이터베이스 구성 매개변수를 -1이 아닌 값으로 설정하십시오.

  이것이 교착 상태를 방지하진 않지만 실행이 다시 재개될 수 있도록 해줍니다. Context 2는 요청된 잠금을 얻을 수 없으므로 결국 구간 복원됩니다. 구간 복원 오류를 처리할 때 Context 2는 세마포어를 릴리스해야 합니다. 일단 세마포어가 릴리스되면, Context 1은 계속 수행하고, Context 2는 해제되어 작업을 재시도합니다.

교착 상태를 방지하는 기법은 위의 예를 중심으로 설명되었지만, 사용자는 이 기법들을 모든 다중 스레드 응용프로그램에 적용할 수 있습니다. 일반적으로, 데이터베이스 관리 프로그램을 보호 자원을 취급할 때처럼 취급하면 다중 스레드 응용프로그램에서 발생할 수 있는 문제점이 발생하지 않을 것입니다.

동시 트랜잭션

때로 응용프로그램이 *동시 트랜잭션*이라는 여러개의 독립적인 연결을 가지고 있으면 유용합니다. 동시 트랜잭션을 사용하여 응용프로그램은 동시에 여러개의 데이터베이스에 연결하고 같은 데이터베이스에 여러개의 연결을 설정할 수 있습니다.

607 페이지의 『다중 스레드 데이터베이스 액세스』에서 설명된 문맥 API를 통해 응용프로그램은 동시 트랜잭션을 사용할 수 있습니다. 응용프로그램에서 작성된 각 문맥은 다른 문맥과는 독립적입니다. 이것은 사용자가 문맥을 작성하고, 그 문맥을 사용하여 데이터베이스에 연결하며, 다른 문맥의 COMMIT이나 ROLLBACK문 수행과 같은 조치에 의해 영향을 받지 않는 데이터베이스에 대하여 SQL문을 수행한다는 것을 의미합니다.

예를 들어, 사용자가 하나의 데이터베이스에 대하여 SQL문을 수행하도록 허용하고, 두번째 데이터베이스에서 수행된 조치의 로그를 보관하는 응용프로그램을 작성한다고 가정하십시오. 이 로그는 최신의 상태를 유지해야 하므로, 로그의 매 갱신 후에 COMMIT문을 발행해야 하지만, 사용자는 SQL문이 로그에 대한 확약의 영

향을 받는 것을 원하지 않습니다. 이것이 동시 트랜잭션을 위한 완벽한 상황입니다. 사용자의 응용프로그램에서, 두 개의 문맥을 작성하십시오. 하나는 사용자의 데이터베이스에 연결되어 모든 사용자 SQL에 사용되고, 다른 하나는 로그 데이터베이스에 연결되어 로그 갱신에 사용됩니다. 이러한 설계로 사용자가 로그 데이터베이스의 변경사항을 확약할 때 사용자의 현재 작업 단위(UOW)에는 영향을 미치지 않습니다.

동시 트랜잭션의 또 다른 이점은 하나의 연결에서 커서에 대한 작업이 구간 복원되면, 다른 연결의 커서에 영향을 미치지 않는다는 것입니다. 하나의 연결에서의 구간 복원 후에, 작업도 완료되고, 다른 연결에서 커서의 위치도 여전히 유지됩니다.

동시 트랜잭션 사용시 잠재적 함정

동시 트랜잭션을 사용하는 응용프로그램은 단일 연결을 사용하는 응용프로그램 작성시에는 발생하지 않는 문제점이 발생할 수 있습니다. 동시 트랜잭션을 사용하여 응용프로그램을 작성할 때, 다음을 주의하십시오.

- 둘 이상의 문맥 간에 데이터베이스 종속성

  응용프로그램에서 각 문맥은 데이터베이스 오브젝트에 대한 잠금과 같은 각자의 데이터베이스 자원 세트를 가지고 있습니다. 이러한 점으로 인해 두 개의 문맥이 하나의 데이터베이스 오브젝트에 액세스하는 경우 교착 상태에 놓이게 될 수 있습니다. 데이터베이스 관리 프로그램은 교착 상태를 발견하고, 문맥 중 하나가 SQLCODE -911을 수신하며, 그 작업 단위(UOW)는 구간 복원됩니다.

- 둘 이상의 문맥 간의 응용프로그램 종속성

  하나의 스레드에서 문맥을 전환하는 것은 문맥 간의 종속 관계를 만드는 것입니다. 문맥에 데이터베이스 종속성도 있는 경우, 교착 상태에 놓일 가능성이 있습니다. 일부 종속성은 데이터베이스 관리 프로그램 외부에 있기 때문에 교착 상태는 발견되지 않고 응용프로그램은 일시 중단 상태가 됩니다.

  이러한 종류의 문제점의 예로, 다음의 응용프로그램을 생각해 봅시다.

  ```
  context 1
  UPDATE TAB1 SET COL = :new_val

  context 2
  SELECT * FROM TAB1
  ```

```
COMMIT

context 1
COMMIT
```

첫번째 문맥이 성공적으로 UPDATE문을 실행했다고 가정하십시오. 갱신은 TAB1의 모든 행에 잠금을 설정했습니다. 이제 Context 2가 TAB1의 모든 행을 선택하려 합니다. 두 문맥이 독립적이므로, Context 2는 Context 1이 차지하고 있는 잠금을 기다립니다. 그러나 Context 1은 Context 2가 실행을 종료할 때까지 잠금을 릴리스할 수 없습니다. 이제 응용프로그램은 교착 상태에 놓이게 되었으나 데이터베이스 관리 프로그램은 Context 1이 Context 2를 기다리고 있음을 모르고 있으므로, 문맥 중 하나의 구간 복원을 강제하지 않습니다. 그래서 응용프로그램이 일시 중단됩니다.

동시 트랜잭션의 교착 상태 방지

데이터베이스 관리 프로그램이 문맥 간의 교착 상태를 발견할 수 없으므로, 교착 상태를 방지하거나 최소한 피할수 있는 방식으로 응용프로그램을 설계하고 코딩해야 합니다. 위의 예에서, 여러 가지 방법으로 교착 상태를 피할 수 있습니다.

- 문맥을 전환하기 전에 보류되어 있는 잠금을 모두 릴리스 하십시오.

  코드를 변경하여 Context 1이 Context 2로 전환하기 전에 확약을 수행하도록 하십시오.

- 한 번에 둘 이상의 문맥에서 해당 오브젝트에 액세스하지 마십시오.

  코드를 변경하여 같은 문맥에서 갱신 및 선택이 모두 이루어지도록 하십시오.

- LOCKTIMEOUT 데이터베이스 구성 매개변수를 -1이 아닌 값으로 설정하십시오.

  이것이 교착 상태를 방지하진 않지만 실행이 다시 재개될 수 있도록 해줍니다. Context 2는 요청된 잠금을 얻을 수 없으므로 결국 구간 복원됩니다. 일단 Context 2가 구간 복원되면, Context 1은 실행을 계속하고, 그러면 잠금이 해제되며, Context 2는 작업을 재시도합니다.

교착 상태를 방지하는 기법은 위의 예를 중심으로 설명되었지만, 사용자는 이 기법들을 모든 동시 트랜잭션 응용프로그램에 적용할 수 있습니다.

X/Open XA 인터페이스 프로그래밍 고려사항

X/Open® XA 인터페이스는 복수의 자원에 대한 변경을 조정하고, 이러한 변경사항의 무결성을 보장하는 개방형 표준입니다. *트랜잭션 처리 모니터*라고 알려져 있는 소프트웨어 모니터들은 일반적으로 XA 인터페이스를 사용하며, DB2가 이 인터페이스를 지원하므로, 하나 이상의 DB2 데이터베이스가 이러한 환경에서 자원으로서 동시에 액세스될 수 있습니다. 데이터베이스 관리 프로그램에서 제공되는 XA 인터페이스 지원의 개념과 구현에 대한 정보는 *관리 안내서: 계획* 책을 참조하십시오. 사용자의 플랫폼이 X/Open XA Interface를 지원하는지 여부를 판별하려면, *응용프로그램 빌드 안내서* 책을 참조하십시오.

트랜잭션 처리시 TP 모니터와는 독립적으로 수행되는 응용프로그램에 비교할 때 다른 모델이 사용되므로, XA 인터페이스를 사용하는 분산 트랜잭션 프로세싱(DTP) 환경에서 운영할 때는 특별한 고려사항이 있습니다. 이 트랜잭션 처리 모델의 특징은 다음과 같습니다.

1. DB2 데이터베이스와 같은 복구 가능한 자원의 여러 유형이 트랜잭션 내에서 수정될 수 있습니다.
2. 자원은 2단계 확약을 사용하여 갱신되어 실행되는 트랜잭션의 무결성을 보장합니다.
3. 응용프로그램은 자원 관리 프로그램이 아닌 TP 모니터 제품으로 트랜잭션의 확약이나 구간 복원 요청을 송신합니다. 예를 들어, CICS 환경에서 응용프로그램은 `EXEC CICS SYNCPOINT`를 발행하여 트랜잭션을 확약할 수 있는데, `EXEC SQL COMMIT`을 DB2로 발행하는 것은 유효하지 않으며 불필요합니다.
4. 트랜잭션을 수행할 수 있는 권한이 TP 모니터와 관련 소프트웨어에 의해 가려지므로, DB2와 같은 자원 관리자는 TP 모니터를 권한이 부여된 단일 사용자로 취급합니다. 예를 들어, CICS 트랜잭션 사용은 CICS에 의해 권한이 부여되어야 하고, 데이터베이스로의 액세스 특권은 CICS 응용프로그램을 호출한 일반 사용자가 아니라 CICS에게 권한 부여받아야 합니다.
5. 복수의 프로그램(트랜잭션)들은 일반적으로 대기행렬에 들어갔다가 데이터베이스 서버(DB2의 관점에서는 단일한 장기 수행 응용프로그램)에서 실행됩니다.

이러한 환경의 고유한 특성으로 DB2는 여기에서 수행되도록 코딩된 응용프로그램에 대하여 특별한 동작 및 요구사항이 있습니다.

- 분산 작업 단위(DUOW) 사전 처리 컴파일러 옵션이나 클라이언트 설정값을 고려하지 않고 작업 단위(UOW) 내에서 복수의 데이터베이스들이 연결되고 갱신될 수 있습니다.
- DISCONNECT문은 허용되지 않으며 시도되는 경우 SQLCODE -30090(SQLSTATE 25000)과 함께 거부됩니다.
- RELEASE문은 트랜잭션이 확약될 때 릴리스할 데이터베이스 연결을 지정하는데 사용될 수 있지만 권장되는 방법은 아닙니다. 연결이 릴리스되면, 후속 트랜잭션들이 SET CONNECTION문을 사용하여 권한을 요구하지 않고 데이터베이스로 연결해야 합니다.
- COMMIT과 ROLLBACK문은 TP 모니터 트랜잭션에 의해 액세스되는 저장 프로시듀어 내에서 허용되지 않습니다.
- 2 단계 확약 흐름이 트랜잭션에 대하여 명시적으로 작동불가능 상태가 될 때 (이들을 XA 인터페이스 용어로 *LOCAL* 트랜잭션이라고 합니다), 단 하나의 데이터베이스만이 이 트랜잭션 내에서 액세스될 수 있습니다. 이 데이터베이스는 SNA 연결성을 사용하여 액세스되는 호스트 또는 AS/400 데이터베이스가 될 수 없습니다. TCP/IP 연결성을 사용하여 OS/390용 DB2 버전 5에 대한 지역 트랜잭션은 지원됩니다.
- LOCAL 트랜잭션은 각 트랜잭션 끝에 SQL COMMIT 또는 SQL ROLLBACK을 발행해야 하는데, 그렇지 않은 경우 트랜잭션은 처리되는 다음 트랜잭션의 일부로 간주됩니다.
- 현재의 데이터베이스 연결들 간의 전환은 SQL CONNECT 또는 SQL SET CONNECTION의 사용을 통해 이루어집니다. 연결에 사용되는 권한은 CONNECT문에 사용자 ID나 암호를 지정하여 변경할 수 없습니다.
- 테이블, 뷰 또는 색인과 같은 데이터베이스 오브젝트가 동적 SQL문에서 완전한 이름이 아닌 경우, 이것은 사용자 ID가 아닌 TP 모니터가 그 아래에서 실행되고 있는 단일 권한 부여 ID를 사용하여 내재적으로 규정됩니다.
- LOCAL이 아닌 트랜잭션에 대하여 DB2 COMMIT 또는 ROLLBACK문의 사용은 거부됩니다. 다음의 코드가 리턴됩니다.
  - 정적 COMMIT에는 SQLCODE -925 (SQLSTATE 2D521)
  - 정적 ROLLBACK에는 SQLCODE -926 (SQLSTATE 2D521)
  - 동적 COMMIT에는 SQLCODE -426 (SQLSTATE 2D528)

– 동적 ROLLBACK에는 SQLCODE -427 (SQLSTATE 2D529)

- COMMIT이나 ROLLBACK을 위한 CLI 요청 역시 거부됩니다.
- Handling database-initiated rollback:

  DTP 환경에서, 예를 들어 시스템 오류나 교착 상태로 인해 전역 트랜잭션의 자신의 분기를 종료하기 위해 RM이 구간 복원을 시작한 경우, 트랜잭션 관리 프로그램이 시작한 동기 시점 요청이 발생할 때까지 동일한 응용프로그램 프로세스로부터 오는 더 이상의 요청은 처리해서는 안됩니다. 여기에는 저장 프로시듀어 내에서 발생한 교착 상태도 포함됩니다. 데이터베이스 관리 프로그램의 경우, 이것은 SQLCODE -918(SQLSTATE 51021)과 함께 모든 후속 SQL 요청을 거부하고, 사용자에게 CICS 환경에서 CICS SYNCPOINT ROLLBACK 명령 사용과 같은 트랜잭션 관리 프로그램의 동기 시점 서비스를 사용하여 전역 트랜잭션을 구간 복원해야 함을 알려줍니다. 어떤 이유에서건, 사용자가 대신 트랜잭션을 확약하기 위해 TM에게 요청한 경우, RM은 TM에게 구간 복원에 대해 알려주고, TM이 다른 RM을 구간 복원하도록 합니다.

- WITH HOLD로 선언된 커서:

  WITH HOLD로 선언된 커서는 CICS 트랜잭션 처리 모니터용 XA/DTP 환경에서 지원됩니다.

  WITH HOLD로 선언된 커서가 지원되지 않는 경우, OPEN문은 SQLCODE -30090(SQLSTATE 25000)과 함께 거부되며 이유 코드는 03입니다.

  WITH HOLD로 지정된 커서가 더 이상 필요 없을 때 명시적으로 닫히도록 하는 것은 트랜잭션의 책임입니다. 그렇지 않은 경우 이 커서들은 다른 트랜잭션에 의해 계승되어 자원의 충돌이나 불필요한 소모를 유발합니다.

- 데이터베이스를 갱신하거나 변경하는 명령문이 2 단계 확약 요청 흐름을 지원하지 않는 데이터베이스에 대해서는 허용되지 않습니다. 예를 들어, DRDA 프로토콜(DRDA2)의 레벨 2가 지원되지 않는 환경에서의 호스트 또는 AS/400 데이터베이스 서버에의 액세스(897 페이지의『DB2 Connect와 다중 사이트 갱신』 참조).
- 데이터베이스가 XA 환경에서 갱신을 지원하는지 여부는 CONNECT문을 발행하여 런타임시에 판별할 수 있습니다. 데이터베이스 갱신이 가능하면 세번째 SQLERRD 토큰에 1 값이 들어 있고, 아니면 2 값이 들어 있습니다.
- 갱신이 제한되면, 다음의 SQL문만이 허용됩니다.

```
CONNECT
DECLARE
DESCRIBE
EXECUTE IMMEDIATE (여기서, 첫번째 토큰 또는 키워드는 SET이고
                   SET CONSTRAINTS가 아님)
OPEN CURSOR
FETCH CURSOR
CLOSE CURSOR
PREPARE (여기서, 첫번째 토큰 또는 키워드는 공백이 아니거나
         왼쪽 괄호가 SET(SET CONSTRAINTS가 아님),
         SELECT, WITH 또는 VALUES)
SELECT...INTO
VALUES...INTO
```

그외 다른 시도는 거부되며 SQLCODE -30090(SQLSTATE 25000)이 수신됩니다.

PREPARE문은 SELECT문을 준비하기 위해서만 사용됩니다. EXECUTE IMMEDIATE문 역시 OS/390용 DB2 Universal Database의 SET SQLID문과 같이 아무런 출력 값을 리턴하지 않는 SQL SET문을 실행하도록 허용됩니다.

- API 제한사항:

  내부적으로 데이터베이스에 확약을 발행하고 2 단계 확약 처리를 바이패스하는 API는 거부되며, SQLCODE -30090(SQLSTATE 25000)이 수신됩니다. 이러한 API 목록은 605 페이지의『다중사이트 갱신 제한사항』의 내용을 참조하십시오. 이 API들은 다중 사이트 갱신(연결 유형 2)에서 지원되지 않습니다.

- 응용프로그램은 단일 스레드여야 합니다.

  다중 스레드 응용프로그램을 개발하기로 하였다면, 하나의 스레드가 SQL을 사용하도록 하거나, 같은 작업 단위(UOW) 안에서 다른 스레드로부터의 SQL문과 얽히지 않도록 다중 프로세스 설계를 사용해야 합니다. 트랜잭션 관리 프로그램이 다중 프로세스나 다중 스레드를 지원하는 경우, 스레드를 직렬화하여 다른 스레드가 시작하기 전에 하나의 스레드가 동기점까지 실행할 수 있도록 해야 합니다. 예는 AIX/CICS의 ***all_operation***의 **XASerialize** 옵션입니다. 이 정보가 들어 있는 AIX/CICS XAD 파일에 대한 세부사항은 *관리 안내서: 계획*의 내용을 참조하십시오.

위의 제한사항이 XA 인터페이스를 사용하는 TP 모니터 환경에서 수행되는 응용프로그램에 적용됨에 주의하십시오. DB2 데이터베이스가 XA 인터페이스와 함께 사용되도록 정의되어 있지 않은 경우, 이러한 제한사항은 적용되지 않지만, 여전히 트랜잭션 코딩시 부주의하게 다음에 실행될 트랜잭션에 영향을 미치는 상태가 되도록 하지 않는 방식으로 코딩하는 것이 필요합니다.

응용프로그램 링크

실행 가능 응용프로그램을 생성하기 위하여, 응용프로그램 오브젝트를 언어 라이브러리, 운영 체제 라이브러리, 일반 데이터베이스 관리 프로그램 라이브러리 및 TP 모니터와 트랜잭션 관리 프로그램 제품 라이브러리와 링크해야 합니다.

네트워크에 걸쳐 있는 대용량의 데이터에 대한 작업

네트워크에 걸쳐 대용량의 데이터를 전달해야 하는 응용프로그램의 성능을 크게 향상시키려면, 207 페이지의 『제7장 저장 프로시듀어』에 설명된 저장 프로시듀어 기법과 *관리 안내서: 구현*에서 설명된 행 블로킹 기법을 결합할 수 있습니다.

네트워크에서 배열, 대용량의 데이터 또는 데이터 패키지 등을 전달하는 응용프로그램은 전달 메카니즘으로 SQLDA 데이터 구조나 호스트 변수를 사용하여 데이터를 블록 단위로 전달할 수 있습니다. 이러한 기법은 구조를 지원하는 호스트 언어에서 특히 강력합니다.

클라이언트 응용프로그램이나 서버 프로시듀어가 네트워크에서 데이터를 전달할 수 있습니다. 다음과 같은 데이터 유형 중 하나를 사용하여 전달될 수 있습니다.
- VARCHAR
- LONG VARCHAR
- CLOB
- BLOB

또한 다음과 같은 그래픽 유형 중 하나를 사용하여 전달될 수 있습니다.
- VARGRAPHIC
- LONG VARGRAPHIC
- DBCLOB

이 주제에 대한 자세한 정보는 91 페이지의 『데이터 유형』의 내용을 참조하십시오.

주: 이 기법을 사용할 때 데이터 변환의 가능성을 반드시 고려하십시오. VARCHAR, LONG VARCHAR 또는 CLOB과 같은 문자열 데이터 유형이나 VARGRAPHIC, LONG VARGRAPHIC 또는 DBCLOB와 같은 그래픽 데이터 유형 중 하나를 갖는 데이터를 전달하고, 응용프로그램 코드 페이지가 데이터베이스 코드 페이지와 같지 않은 경우, 비 문자 데이터가 문자 데이터처럼 변환될 수 있습니다. 문자 변환을 방지하려면, 데이터를 BLOB 데이터 유형의 변수에 전달해야 합니다.

데이터 변환이 발생하는 방법 및 시점에 대한 정보는 573 페이지의 『다른 코드 페이지 간의 변환』의 내용을 참조하십시오.

제18장 파티션된 환경에서의 프로그래밍 고려사항

성능 향상 621
FOR READ ONLY 커서 사용 621
방향이 지정된 DSS와 지역 바이패스 사용 621
방향이 지정된 DSS 622
지역 바이패스 사용 623
버퍼 삽입 사용 623
버퍼 삽입 사용시 고려사항 627
버퍼 삽입 사용시 제한 사항 630
예: 대용량의 데이터 추출(largevol.c) . . 630
테스트 환경 작성 636
오류 처리시 고려사항 636
심각한 오류. 637
병합된 복수 SQLCA 구조. 637
오류를 리턴하는 파티션 식별 639
디버깅 639
루핑 또는 일시정지한 응용프로그램 진단 639

성능 향상

파티션된 환경이 제공하는 성능상의 이점을 활용하려면, 특별한 프로그래밍 기법 사용을 고려해야 합니다. 예를 들어, 사용자의 응용프로그램이 둘 이상의 데이터베이스 관리 프로그램 구획으로부터 DB2 데이터에 액세스하는 경우, 그 안에 들어 있는 정보를 고려해야 합니다. 파티션된 환경에 대한 개요는 *관리 안내서* 및 *SQL 참조서* 책을 참조하십시오.

FOR READ ONLY 커서 사용

읽기만을 위한 커서를 선언하려는 경우, OPEN CURSOR 선언에 FOR READ ONLY 또는 FOR FETCH ONLY를 포함시키십시오. FOR READ ONLY와 FOR FETCH ONLY는 동일한 명령문입니다. FOR READ ONLY 커서를 사용하여 조정자 파티션은 한 번에 여러 행을 검색함으로써 후속 FETCH문의 성능을 크게 향상시킬 수 있습니다. 명시적으로 커서를 FOR READ ONLY로 선언하지 않으면, 조정자 파티션은 이들을 갱신 가능한 커서로 취급합니다. 갱신 가능한 커서는 조정자 파티션이 FETCH 한 번에 한 행만을 검색하도록 하기 때문에 비용 소모가 큽니다.

방향이 지정된 DSS와 지역 바이패스 사용

온라인 트랜잭션 처리 응용프로그램을 최적화하기 위해, 사용자는 모든 데이터 파티션에서의 처리를 요구하는 단순한 SQL문을 피하려할 수 있습니다. 사용자는 SQL

© Copyright IBM Corp. 1993, 2000

문이 하나의 파티션에서 데이터를 검색할 수 있도록 응용프로그램을 설계해야 합니다. 이 기법은 조정자 파티션이 연관된 파티션 하나 또는 모두와 통신하는 데 소모되는 비용을 줄일 수 있습니다.

방향이 지정된 DSS

분산 하위 섹션(DSS)는 하위 섹션을 병렬 조회에 대해 몇 가지 작업을 수행해야 하는 데이터베이스 파티션으로 보내는 조치입니다. 이는 또한 하위 섹션을 OLTP 환경에 있는 변수의 값과 같은 특정 값을 호출하여 시작하는 방법을 설명합니다. *방향이 지정된 DSS*는 조회를 단일 파티션으로 지정하기 위해 테이블 파티션 키를 사용합니다. 사용자 응용프로그램에서 이러한 유형의 조회를 사용하여 모든 노드로 조회 브로드캐스트를 하는 데 필요한 조정자 파티션의 오버헤드를 피하십시오.

방향이 지정된 DSS를 활용할 수 있는 예제 SELECT문의 일부는 다음과 같습니다.

```
SELECT ... FROM t1
  WHERE PARTKEY=:hostvar
```

조정자 파티션이 조회를 수신하면, 이것은 어떤 파티션이 *:hostvar*의 데이터 부속 집합을 가지고 있는지를 판별하고, 조회를 해당 파티션으로 지정합니다.

방향이 지정된 DSS를 사용하여 응용프로그램을 최적화하려면, 복잡한 조회를 여러개의 단순한 조회로 나누십시오. 예를 들어, 다음 조회에서 조정자 파티션은 파티션 키를 여러 값과 대응합니다. 조회를 만족시키는 데이터가 여러 파티션에 존재하므로, 조정자 파티션은 이 조회를 모든 파티션으로 브로드캐스트합니다.

```
SELECT ... FROM t1
  WHERE PARTKEY IN (:hostvar1, :hostvar2)
```

대신 이 조회를 하나의 호스트 변수를 갖는 여러개의 SELECT문으로 나누거나 UNION을 갖는 하나의 SELECT문을 사용하여 같은 결과를 얻으십시오. 조정자 파티션은 더 단순한 SELECT문을 활용하여 필요한 파티션과 통신하기 위해 방향이 지정된 DSS를 사용합니다. 최적화된 조회는 다음과 같습니다.

```
SELECT ... AS res1 FROM t1
  WHERE PARTKEY=:hostvar1
  UNION
SELECT ... AS res2 FROM t1
  WHERE PARTKEY=:hostvar2
```

위의 기법은 UNION의 select의 수가 파티션 수보다 훨씬 적을 때에만 성능을 향상시킬 수 있습니다.

지역 바이패스 사용

방향이 지정된 DSS 조회의 특별한 형태가 조정자 파티션에만 저장되어 있는 데이터에 액세스합니다. 조정자 파티션이 다른 파티션과 통신할 필요가 없이 조회를 완료하기 때문에 이것을 *지역 바이패스*라고 합니다.

지역 바이패스는 가능할 때마다 자동으로 작동가능해지는데, 해당 트랜잭션에 대한 데이터가 들어 있는 파티션으로 트랜잭션을 라우팅함으로써 그 사용을 증가시킬 수 있습니다. 이를 수행하는 한 가지 기법은 원격 클라이언트가 각 파티션으로의 연결을 유지보수하도록 하는 것입니다. 그러면 트랜잭션은 입력 파티션 키에 근거하여 올바른 연결을 사용할 수 있습니다. 또 다른 기법은 파티션별로 트랜잭션을 그룹화하고, 각 파티션에 별도의 응용프로그램 서버를 두는 것입니다.

트랜잭션 데이터가 상주하는 파티션 수를 판별하려면, `sqlugrpn` API(행 파티션 수 구하기)를 사용할 수 있습니다. 이 API를 사용하여 응용프로그램은 파티션 키가 주어졌을 때, 행의 파티션 수를 효율적으로 계산할 수 있습니다. `sqlugrpn` API에 대한 자세한 정보는 *Administrative API Reference* 책을 참조하십시오.

또 다른 방법은 `db2atld` 유틸리티를 사용하여 파티션 번호별로 입력 데이터를 나누고 각 파티션에 대하여 응용프로그램 사본을 수행하는 것입니다. `db2atld` 유틸리티에 대한 자세한 정보는 *Command Reference* 책을 참조하십시오.

버퍼 삽입 사용

버퍼 삽입은 삽입된 행을 버퍼에 넣기 위해 테이블 대기행렬을 활용하는 삽입 명령문으로, 이것을 통해 성능이 크게 향상될 수 있습니다. 버퍼 삽입을 사용하여 응용프로그램은 준비되거나 INSERT BUF 옵션과 바인드되어야 합니다.

버퍼 삽입 결과 삽입을 수행하는 응용프로그램에서 근본적으로 성능이 향상될 수 있습니다. 일반적으로 한 루프 내에서 많은 행을 삽입시키기 위해 하나의 삽입 명령문이 사용되고 다른 데이터베이스 수정 명령문이 사용되지 않고, 데이터 소스가 INSERT문의 VALUES절인 응용프로그램에서 버퍼 삽입을 사용할 수 있습니다.

일반적으로 INSERT문은 연속되는 루프의 실행 중에 그 값이 바뀌는 하나 이상의 호스트 변수를 참조합니다. VALUES절은 하나의 행 또는 복수 행을 지정할 수 있습니다.

일반적인 결정 지원 응용프로그램에서는 새로운 데이터의 로드 및 주기적 삽입을 필요로 합니다. 이 데이터는 수백, 수천 행으로 구성되었을 수 있습니다. 테이블을 로드할 때 버퍼 삽입을 사용하기 위해 응용프로그램을 준비하고 바인드할 수 있습니다.

응용프로그램이 버퍼 삽입을 사용하도록 하려면, PREP 명령을 사용하여 응용프로그램 소스 파일을 처리하거나 결과 바인드 파일에서 BIND 명령을 사용하십시오. 두 가지 경우 모두에서 INSERT BUF 옵션을 지정해야 합니다. 응용프로그램 바인딩에 대한 자세한 정보는 65 페이지의 『바인딩』의 내용을 참조하십시오. 응용프로그램 준비에 대한 자세한 정보는 57 페이지의 『소스 파일 작성 및 준비』의 내용을 참조하십시오.

주: 버퍼 삽입시 다음과 같은 단계가 수행됩니다.

1. 데이터베이스 관리 프로그램이 테이블이 상주하는 각 노드에 대하여 4 KB 버퍼를 엽니다.
2. 응용프로그램에 의해 발행된 VALUES절을 갖는 INSERT문을 사용하여 행이 적절한 버퍼에 들어갈 수 있게 합니다.
3. 데이터베이스 관리 프로그램은 응용프로그램에 제어를 리턴합니다.
4. 버퍼가 가득 차거나 일부만 찬 버퍼의 행을 송신하도록 하는 이벤트가 발생한 경우, 버퍼의 행이 파티션으로 송신됩니다. 일부 찬 버퍼는 다음 중 하나가 발생하는 경우 비워집니다.
   - 응용프로그램은 암시적으로 또는 응용프로그램 종료를 통해 명시적으로 COMMIT이나 ROLLBACK을 지정합니다.
   - 응용프로그램은 저장점을 발생하도록 하는 다른 명령문을 지정합니다. OPEN, FETCH 및 CLOSE 커서 명령문은 저장점이 발생하도록 하지도 않고, 열려 있는 버퍼 삽입을 닫지도 않습니다.

     다음의 SQL문은 열려 있는 버퍼 삽입을 닫습니다.
     - BEGIN COMPOUND SQL
     - COMMIT

- DDL
- DELETE
- END COMPOUND SQL
- EXECUTE IMMEDIATE
- GRANT
- 다른 테이블로 INSERT
- 이름으로 버퍼 삽입을 수행하는 동일한 동적 명령문의 PREPARE
- REDISTRIBUTE NODEGROUP
- RELEASE SAVEPOINT
- REORG
- REVOKE
- ROLLBACK
- ROLLBACK TO SAVEPOINT
- RUNSTATS
- SAVEPOINT
- SELECT INTO
- UPDATE
- 다른 명령문의 실행, 그러나 버퍼 INSERT의 또 다른 루핑 실행은 아님.
- 응용프로그램의 종료

다음의 API는 열려 있는 버퍼 삽입을 닫습니다.
- BIND (API)
- REBIND (API)
- RUNSTATS (API)
- REORG (API)
- REDISTRIBUTE (API)

다른 명령문들이 버퍼 삽입을 닫는 이러한 상황에서, 조정자 노드는 모든 노드가 버퍼를 수신하고 행이 삽입될 때까지 기다립니다. 그리고나서 모든 행이 성공적으로 삽입되면, 다른 명령문(버퍼 삽입을 닫는 명령문)을 실행합니다. 추가 세부사항은 627 페이지의 『버퍼 삽입 사용시 고려사항』의 내용을 참조하십시오.

파티션된 환경에서 표준 인터페이스는 버퍼 삽입 없이 한 번에 한 행을 로드하며, 응용프로그램이 파티션 중 하나에서 지역적으로 수행된다는 가정 하에 다음과 같은 단계를 수행합니다.

1. 조정자 노드는 이 행을 같은 노드의 데이터베이스 관리 프로그램으로 전달합니다.
2. 데이터베이스 관리 프로그램은 간접 해싱을 사용하여 이 행이 위치해야 할 파티션을 판별하십시오.
   - 목표 파티션이 행을 수신합니다.
   - 목표 파티션이 지역적으로 행을 삽입합니다.
   - 목표 파티션이 조정자 노드로 응답을 송신합니다.
3. 조정자 노드가 목표 파티션으로부터 응답을 받습니다.
4. 조정자 노드는 응용프로그램으로 응답을 줍니다.

   삽입은 응용프로그램이 COMMIT을 지정하기 전까지는 확약되지 않습니다.
5. VALUES절이 들어 있는 INSERT문은 무엇이든 행 수나 행의 요소 수에 상관없이 버퍼 삽입의 후보입니다. 즉, 요소는 상수, 특수 레지스터, 호스트 변수, 표현식, 함수 등이 될 수 있습니다.

VALUES절을 갖는 INSERT문의 경우, DB2 SQL 컴파일러는 문법, 성능 또는 구현시 고려사항에 근거하여 삽입을 버퍼에 넣을 수 없습니다. INSERT BUF 옵션을 사용하여 응용프로그램을 준비하거나 바인드하는 경우, 버퍼 삽입에 종속되지 않도록 하십시오. 이것은 다음을 의미합니다.

- 오류는 버퍼 삽입에 비동기적으로 보고되거나, 일반적인 삽입에 동기적으로 보고될 수 있습니다. 비동기적으로 보고되는 경우, 삽입 오류는 버퍼 내 후속 삽입 시 또는 버퍼를 닫는 다른 명령문에서 보고됩니다. 오류를 보고하는 명령문은 실행되지 않습니다. 예를 들어 버퍼 삽입 루프를 닫기 위해 COMMIT문을 사용하는 것을 생각해 봅시다. 이전 삽입에 중복 키가 있으므로 확약은 SQLCODE -803(SQLSTATE 23505)을 보고합니다. 이러한 시나리오에서 확약은 실행되지 않습니다. 예를 들어 사용자의 응용프로그램이 버퍼 삽입 루프를 입력하기 전에 수행된 일부 갱신 내용을 확약하고자 하는 경우, COMMIT문을 다시 발행해야 합니다.

- 삽입된 행은 버퍼 삽입 없는 커서를 사용하여 SELECT문을 통해 즉시 볼 수 있습니다. 버퍼 삽입을 사용하면 행을 즉시 볼 수는 없습니다. INSERT BUF 옵션을 사용하여 응용프로그램을 사전 처리 컴파일하거나 바인드하는 경우, 응용프로그램이 이러한 커서 선택 행에 종속되지 않도록 작성하십시오.

버퍼 삽입은 다음과 같은 성능상의 이점이 있습니다.

- 목표 파티션이 수신하는 각 버퍼에 대하여 단 하나의 메시지가 목표 파티션에서 조정자 노드로 송신됩니다.
- 버퍼에는 행의 크기가 작은 경우 특히 많은 수의 행들이 들어 있을 수 있습니다.
- 조정자 노드가 새로운 행을 수신하는 동안, 파티션에서 삽입이 일어나는 병렬 처리가 발생합니다.

INSERT BUF와 바인드된 응용프로그램은 버퍼 삽입을 닫는 명령문이나 API가 발행되기 전에 VALUES절이 있는 동일한 INSERT문이 반복되도록 작성되어야 합니다.

주: 버퍼 삽입이 트랜잭션 로그를 채우지 않도록 방지하기 위해 주기적인 확약을 해야 합니다.

버퍼 삽입 사용시 고려사항

버퍼 삽입은 응용프로그램에 영향을 미칠 수 있는 기능을 가지고 있습니다. 이 기능은 버퍼 삽입의 비동기적 특성으로부터 기인합니다. 행의 파티션 키에 기초하여 각 삽입된 행은 올바른 파티션에 지정된 버퍼에 놓이게 됩니다. 이 버퍼들이 가득 차게 되면 목적지 파티션으로 송신되거나, 이벤트를 발생시켜 비우게 됩니다. 응용프로그램을 설계하고 코딩할 때 다음과 같은 점들을 숙지하고 고려해야 합니다.

- INSERT문이 실행될 때 삽입된 행의 일부 오류 조건들이 보고되지 않습니다. 이들은 INSERT(또는 다른 테이블로의 INSERT)가 아닌 다른 명령문, 즉 DELETE, UPDATE, COMMIT 또는 ROLLBACK과 같은 첫번째 명령문이 실행될 때 나중에 보고됩니다. 버퍼 삽입 명령문을 닫는 명령문이나 API는 오류 보고를 볼 수 있습니다. 또한 삽입 자체의 호출의 경우 이전에 삽입된 행의 오류를 볼 수도 있습니다. 또한 버퍼 삽입 오류가 UPDATE나 COMMIT과 같은 다른 명령문에 의해 보고되는 경우, DB2는 이 명령문 실행을 시도하지 않습니다.

- *행 그룹* 삽입 중에 탐지된 오류로 이 그룹의 모든 행들이 백 아웃됩니다. 행 그룹은 버퍼 삽입 명령문 실행을 통해 삽입된 모든 행으로 정의됩니다.
  - 작업 단위 시작부터,
  - 명령문이 동적인 경우 준비된 이후로 또는
  - 이전의 다른 갱신 명령문 실행 이후로 버퍼 삽입을 닫거나 비우는 명령문들의 목록은 623 페이지의 『버퍼 삽입 사용』의 내용을 참조하십시오.
- SELECT가 커서를 사용하여 실행되는 경우, 같은 응용프로그램에 의해 INSERT 다음에 발행된 SELECT문에는 삽입된 행이 즉시 보이지 않을 수 있습니다.

버퍼 INSERT문은 열려 있거나 닫혀 있습니다. 이 명령을 처음 호출하면 버퍼 INSERT를 열고, 행은 적절한 버퍼에 추가되며, 제어는 응용프로그램으로 리턴됩니다. 후속 호출은 행을 버퍼에 추가하고 명령문은 열린 상태로 둡니다. 이 명령문이 열려 있는 동안 버퍼는 행들이 목표 테이블의 파티션으로 삽입되는 목적지 파티션으로 송신될 수 있습니다. 버퍼 삽입을 닫는 명령문이나 API가 *다른* 버퍼 INSERT문의 호출을 포함하여 버퍼 INSERT문이 열려 있는 동안 호출되었거나, PREPARE문이 열려 있는 버퍼 INSERT문에 대하여 발행된 경우, 열기 명령문은 새로운 요청이 처리되기 전에 닫힙니다. 버퍼 INSERT문이 닫힌 경우, 나머지 버퍼는 비워집니다. 그리고나서 행은 목표 파티션으로 송신되고 삽입됩니다. 모든 버퍼가 송신되고 모든 행이 삽입된 후에야 새로운 요청이 처리를 시작합니다.

INSERT문을 닫는 중에 오류가 탐지된 경우, 새로운 요청에 대한 SQLCA가 오류 서술로 채워지며, 새로운 요청은 수행되지 않습니다. 또한 *INSERT문이 열린 이후* 버퍼 INSERT문을 통해 삽입된 행 그룹 전체가 데이터베이스로부터 제거됩니다. 응용프로그램 상태는 발견된 특정 오류에 대하여 정의됩니다. 예를 들면 다음과 같습니다.

- 오류가 교착 상태인 경우, 버퍼 삽입 절이 열리기 전에 이루어진 변경을 포함하여 트랜잭션이 구간 복원됩니다.
- 오류가 고유 키 위반인 경우, 데이터베이스의 상태는 명령문이 열리기 이전과 동일합니다. 트랜잭션은 활동중인 상태로 남아 있으며, 명령문이 열리기 이전의 변경 내용은 영향을 받지 않습니다.

예를 들어, 버퍼 삽입 옵션과 바인드 된 다음과 같은 응용프로그램을 생각해 봅시다.

```
EXEC SQL UPDATE t1 SET COMMENT='about to start inserts';
DO UNTIL EOF OR SQLCODE < 0;
  READ VALUE OF hv1 FROM A FILE;
  EXEC SQL INSERT INTO t2 VALUES (:hv1);
  IF 1000 INSERTS DONE, THEN DO
     EXEC SQL INSERT INTO t3 VALUES ('another 1000 done');
     RESET COUNTER;
END;
END;
EXEC SQL COMMIT;
```

예를 들어, 파일에 8,000개의 값이 들어 있으나 고유 키 위반 등으로 3,258 값이 적절치 않다고 가정하십시오. 1,000개의 삽입이 각각 다른 SQL문을 실행하고, 이는 INSERT INTO t2문을 닫습니다. 1,000개의 삽입으로 이루어진 네 번째 그룹에서 3,258 값에 대한 오류가 탐지되었습니다. 이것은 다른 값이 더 삽입된 후에 (반드시 다음 삽입은 아님) 탐지될 수 있습니다. 이러한 상황에서 오류 코드가 INSERT INTO t2문에 대하여 리턴됩니다.

또한 오류는 테이블 t3에서 삽입이 시도될 때 탐지될 수도 있는데, 이것은 INSERT INTO t2문을 닫습니다. 이러한 상황에서 오류는 테이블 t2에 적용된다고 하더라도 오류 코드는 INSERT INTO t3문에 대하여 리턴됩니다.

대신 삽입할 행이 3,900개 있다고 가정하십시오. 행 번호 3,258에 대한 오류 보고를 받기 전에, 응용프로그램은 루프를 빠져나와 COMMIT 발행을 시도할 수 있습니다. 고유 키 위반 리턴 코드가 COMMIT문에 대하여 발행되고, COMMIT은 수행되지 않습니다. 응용프로그램이 지금까지 데이터베이스에 있는 3000행을 COMMIT하려고 하는 경우(EXEC SQL INSERT INTO t3 ...의 최종 실행이 이들 3000 행에 대한 저장점을 종료함), COMMIT은 다시 발행되어야 합니다. ROLLBACK에서도 비슷한 고려사항이 마찬가지로 적용됩니다.

주: 버퍼 삽입 사용시, 테이블이 불확실한 상태에 있지 않도록 하려면 리턴된 SQLCODES를 주의깊게 모니터해야 합니다. 예를 들어, 위의 예에서 THEN DO문에서 SQLCODE < 0절을 제거한 경우, 테이블은 불분명한 행 수를 포함한 채 종료할 수 있습니다.

버퍼 삽입 사용시 제한 사항

다음의 제한 사항이 적용됩니다.

- 버퍼 삽입을 활용하려는 응용프로그램의 경우 다음 중 하나가 참이어야 합니다.
  - 응용프로그램은 PREP을 통해 준비되거나 BIND 명령을 사용하여 바인드되고, INSERT BUF 옵션이 지정되어야 합니다.
  - 응용프로그램은 BIND나 PREP API를 사용하여 바인드되어야 하고, SQL_INSERT_BUF 옵션이 지정되어야 합니다.
- VALUES절이 있는 INSERT문의 명시적 또는 암시적 컬럼 목록에 long 필드나 LOBS가 포함되어 있는 경우, INSERT BUF 옵션은 이 명령문에 대해 무시되고, 버퍼 삽입이 아닌 일반적인 삽입 절이 수행됩니다. 이것은 오류 조건이 아니며, 오류나 경고 메시지가 발행되지 않습니다.
- fullselect가 있는 INSERT는 INSERT BUF에 의해 영향을 받지 않습니다. 버퍼 INSERT는 이러한 유형의 INSERT의 성능을 향상시키지 못합니다.
- 버퍼 삽입은 EXECUTE IMMEDIATE문을 통해 수행되기 때문에, CLP 발행 삽입이 아닌 응용프로그램에서만 사용할 수 있습니다.

응용프로그램은 지원되는 클라이언트 플랫폼이라면 어디에서든 수행될 수 있습니다.

예: 대용량의 데이터 추출(largevol.c)

DB2 Universal Database가 병렬 조회 처리에 뛰어난 기능을 제공하지만, 응용프로그램이나 EXPORT 명령에 있어서 단일 연결점은 대용량의 데이터를 추출할 때 병목이 될 수 있습니다. 이것은 데이터베이스 관리 프로그램에서 응용프로그램으로 데이터를 전달하는 것이 하나의 노드, 즉 보통 단일 프로세서에서 실행되는 CPU 집약적인 프로세스이기 때문입니다.

DB2 Universal Database에서는 이러한 병목을 방지할 수 있는 몇 가지 방법을 제공하여, 추출된 데이터 용량이 프로세서의 수를 증가시키면서 시간 단위 별로 커지도록 합니다. 다음의 예에서는 이들 방법의 근간이 되는 개념을 설명합니다.

사용자에게 20개의 노드에 저장되어 있는 EMPLOYEE라는 테이블이 있고, 정당한 부서에서 근무하는(즉 WORKDEPT는 NULL이 아님) 모든 직원들의 우편 목록(FIRSTNME, LASTNAME, JOB)을 생성한다고 가정하십시오.

다음의 조회는 각 노드에서 병렬로 수행되고 하나의 노드, 즉 조정자 노등에서 전체 응답 세트를 생성합니다.

```
SELECT FIRSTNME, LASTNAME, JOB FROM EMPLOYEE WHERE WORKDEPT IS NOT NULL
```

그러나 다음의 조회는 데이터베이스의 각 파티션에서 수행될 수 있습니다. 즉, 5개의 파티션이 있는 경우 각 파티션에 하나씩 5개의 별도의 조회가 필요합니다. 각 조회는 그 기록이 조회가 수행되는 특정 구획에 있는 모든 직원 이름 세트를 생성합니다. 각 지역 결과 세트는 파일로 경로 재지정될 수 있습니다. 그리고나서 결과 세트들은 하나의 결과 세트로 병합되어야 합니다.

AIX에서는 네트워크 파일 시스템(NFS) 파일들의 특성을 사용하여 병합을 자동화할 수 있습니다. 모든 파티션이 자기들의 응답 세트를 NFS 마운트 상의 동일한 파일로 경로를 지정하면, 결과들이 병합됩니다. 대용량의 버퍼로 응답을 블로킹하지 않고 NFS를 사용하는 경우 성능이 저하됨에 주의하십시오.

```
SELECT FIRSTNME, LASTNAME, JOB FROM EMPLOYEE WHERE WORKDEPT IS NOT NULL
                  AND NODENUMBER(NAME) = CURRENT NODE
```

결과는 지역 파일에 저장될 수도 있고(즉 최종 결과가 20개의 파일에 담겨 있으며 각각의 파일에 전체 응답 세트의 일부가 들어 있음), 하나의 NFS 마운트 파일에 저장될 수도 있습니다.

다음의 예에서는 두 번째 방법을 사용하여 결과가 20개의 노드에 걸쳐 마운트되어 있는 NFS인 하나의 파일에 들어 있도록 합니다. NFS 잠금 메카니즘을 통해 다른 파티션으로부터 결과 파일에 기록하는 순서가 체계화됩니다. 이 예는 NFS 파일 시스템이 설치된 AIX 플랫폼에서 수행됨에 주의하십시오.

```
#define _POSIX_SOURCE
#define INCL_32

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sqlenv.h>
#include <errno.h>
#include <sys/access.h>
#include <sys/flock.h>
#include <unistd.h>

#define BUF_SIZE 1500000  /* Local buffer to store the fetched records */
#define MAX_RECORD_SIZE 80 /* >= size of one written record */

int main(int argc, char *argv[]) {
```

```
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
   char dbname[10];  /* Database name (argument of the program) */
   char userid[9];
   char passwd[19];
   char first_name[21];
   char last_name[21];
   char job_code[11];
EXEC SQL END DECLARE SECTION;

   struct flock unlock ;  /* structures and variables for handling */
   struct flock lock ;  /* the NFS locking mechanism */
   int lock_command ;
   int lock_rc ;
   int iFileHandle ;  /* output file */
   int iOpenOptions = 0 ;
   int iPermissions ;
   char * file_buf ;  /* pointer to the buffer where the fetched
                         records are accumulated */
   char * write_ptr ;  /* position where the next record is written */
   int buffer_len = 0 ;  /* length of used portion of the buffer */

/* Initialization */

   lock.l_type = F_WRLCK;  /* An exclusive write lock request */
   lock.l_start = 0;  /* To lock the entire file */
   lock.l_whence = SEEK_SET;
   lock.l_len = 0;
   unlock.l_type = F_UNLCK;  /* An release lock request */
   unlock.l_start = 0;  /* To unlock the entire file */
   unlock.l_whence = SEEK_SET;
   unlock.l_len = 0;
   lock_command = F_SETLKW;  /* Set the lock */
   iOpenOptions = O_CREAT;  /* Create the file if not exist */
   iOpenOptions |= O_WRONLY;  /* Open for writing only */

/* Connect to the database */

   if (argc == 3) {
      strcpy( dbname, argv[2] ); /* get database name from the argument */
      EXEC SQL CONNECT TO :dbname IN SHARE MODE ;
      if ( SQLCODE != 0 ) {
         printf( "Error: CONNECT TO the database failed. SQLCODE = %ld\n",
                SQLCODE );
          exit(1);
      }
   }
   else if ( argc == 5 ) {
     strcpy( dbname, argv[2] ); /* get  database name from the argument */
     strcpy (userid, argv[3]);
     strcpy (passwd, argv[4]);
     EXEC SQL CONNECT TO :dbname IN SHARE MODE USER :userid USING :passwd;
     if ( SQLCODE != 0 ) {
        printf( "Error: CONNECT TO the database failed. SQLCODE = %ld\n",
                SQLCODE );
          exit( 1 );
     }
   }
   else {
       printf ("\nUSAGE: largevol txt_file database [userid passwd]\n\n");
       exit( 1 ) ;
   } /* endif */
```

```
  /* Open the input file with the specified access permissions */

 if ( ( iFileHandle = open(argv[1], iOpenOptions, 0666 ) ) == -1 ) {
   printf( "Error: Could not open %s.\n", argv[2] ) ;
   exit( 2 ) ;
}

/* Set up error and end of table escapes */

EXEC SQL WHENEVER SQLERROR GO TO ext ;
EXEC SQL WHENEVER NOT FOUND GO TO cls ;

/* Declare and open the cursor */

EXEC SQL DECLARE c1 CURSOR FOR
         SELECT firstnme, lastname, job FROM employee
         WHERE workdept IS NOT NULL
         AND NODENUMBER(lastname) = CURRENT NODE;
EXEC SQL OPEN c1 ;

/* Set up the temporary buffer for storing the fetched result */

if ( ( file_buf = ( char * ) malloc( BUF_SIZE ) ) == NULL ) {
   printf( "Error: Allocation of buffer failed.\n" ) ;
   exit( 3 ) ;
}
memset( file_buf, 0, BUF_SIZE ) ; /* reset the buffer */
buffer_len = 0 ;  /* reset the buffer length */
write_ptr = file_buf ;  /* reset the write pointer */
/* For each fetched record perform the following    */
/*  - insert it into the buffer following the       */
/*    previously stored record                      */
/*  - check if there is still enough space in  the  */
/*    buffer for the next record and lock/write/    */
/*    unlock the file and initialize the buffer     */
/*    if not                                        */

do {
   EXEC SQL FETCH c1 INTO :first_name, :last_name, :job_code;
    buffer_len += sprintf( write_ptr, "%s %s %s\n",
                           first_name, last_name, job_code );
    buffer_len = strlen( file_buf ) ;
   /* Write the content of the buffer to the file if */
   /* the buffer reaches the limit                   */
   if ( buffer_len >= ( BUF_SIZE - MAX_RECORD_SIZE ) ) {
    /*  get excl. write lock */
    lock_rc = fcntl( iFileHandle, lock_command, &lock );
             if ( lock_rc != 0 ) goto file_lock_err;
             /*  position at the end of file */
             lock_rc = lseek( iFileHandle, 0, SEEK_END );
            if ( lock_rc < 0 ) goto file_seek_err;
            /* write the buffer */
            lock_rc = write( iFileHandle,
                           ( void * ) file_buf, buffer_len );
            if ( lock_rc < 0 ) goto file_write_err;
              /* release the lock */
             lock_rc = fcntl( iFileHandle, lock_command, &unlock );
             if ( lock_rc != 0 ) goto file_unlock_err;
             file_buf[0] = '\0' ;  /* reset the buffer */
              buffer_len = 0 ;  /* reset the buffer length */
              write_ptr = file_buf ;  /* reset the write pointer */
   }
   else {
```

```
            write_ptr = file_buf + buffer_len ;  /* next write position */
         }
      } while (1) ;

cls:
    /* Write the last piece of data out to the file */
    if (buffer_len > 0) {
       lock_rc = fcntl(iFileHandle, lock_command, &lock);
       if (lock_rc != 0) goto file_lock_err;
      lock_rc = lseek(iFileHandle, 0, SEEK_END);
       if (lock_rc < 0) goto file_seek_err;
       lock_rc = write(iFileHandle, (void *)file_buf, buffer_len);
       if (lock_rc < 0) goto file_write_err;
       lock_rc = fcntl(iFileHandle, lock_command, &unlock);
      if (lock_rc != 0) goto file_unlock_err;
    }
    free(file_buf);
        close(iFileHandle);
    EXEC SQL CLOSE c1;
    exit (0);
 ext:
    if ( SQLCODE != 0 )
       printf( "Error:  SQLCODE = %ld.\n", SQLCODE );
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL CONNECT RESET;
    if ( SQLCODE != 0 ) {
       printf( "CONNECT RESET Error:  SQLCODE = %ld\n", SQLCODE );
       exit(4);
    }
    exit (5);
 file_lock_err:
    printf("Error: file lock error = %ld.\n",lock_rc);
      /* unconditional unlock of the file */
    fcntl(iFileHandle, lock_command, &unlock);
    exit(6);
file_seek_err:
    printf("Error: file seek error = %ld.\n",lock_rc);
      /* unconditional unlock of the file */
    fcntl(iFileHandle, lock_command, &unlock);
   exit(7);
file_write_err:
    printf("Error: file write error = %ld.\n",lock_rc);
      /* unconditional unlock of the file */
    fcntl(iFileHandle, lock_command, &unlock);
    exit(8);
file_unlock_err:
    printf("Error: file unlock error = %ld.\n",lock_rc);
      /* unconditional unlock of the file */
   fcntl(iFileHandle, lock_command, &unlock);
    exit(9);
}
```

이 방법은 단일 테이블로부터의 선택에만 적용되는 것이 아니라 더 복잡한 조회에도 적용될 수 있습니다. 그러나 조회에 비할당 조작이 필요한 경우(즉, Explain이 조정자 부속 절 이외의 둘 이상의 부속 절을 보여주는 경우), 조회가 모든 파티션에서 병렬로 수행될 때, 이것은 일부 파티션에서 너무 많은 프로세스가 발생할 수 있습니다. 이러한 경우, 조회 결과를 필요에 따라 파티션에서 임시 테이블 TEMP에 조회 결과를 저장한 후, TEMP로부터 병렬로 최종 추출을 수행합니다.

선택된 작업 분류에 대해서 모든 직원을 추출하고자 하는 경우, 다음과 같이 컬럼 이름, FIRSTNME, LASTNAME 및 JOB을 사용하여 TEMP 테이블을 정의할 수 있습니다.

```
INSERT INTO TEMP
SELECT FIRSTNME, LASTNAME, JOB FROM EMPLOYEE WHERE WORKDEPT IS NOT NULL
       AND EMPNO NOT IN (SELECT EMPNO FROM EMP_ACT WHERE
                         EMPNO<200)
```

그리고나서 TEMP에서 병렬 추출을 실행합니다.

TEMP 테이블을 정의할 때, 다음을 고려하십시오.

- 조회가 총계 GROUP BY를 지정하는 경우, TEMP의 파티션 키를 GROUP BY 컬럼의 부속 집합으로 정의해야 합니다.
- TEMP 테이블의 파티션 키는 테이블이 정의된 구획에 동등하게 분산되도록 충분한 기본 행수(즉, 응답 세트 값의 수)를 가져야 합니다.
- NOT LOGGED INITIALLY 속성을 갖는 TEMP 테이블을 작성한 후, 테이블을 작성한 작업 단위를 COMMIT하여 확보된 카날로그 잠금이 있으면 해제하십시오.
- TEMP 테이블을 사용할 때, 하나의 작업 단위에서 다음과 같은 명령을 발행해야 합니다.
  1. ALTER TABLE TEMP ACTIVATE NOT LOGGED INITIALLY WITH EMPTY TABLE (TEMP 테이블을 비우고 로그 오프 하기 위해).
  2. INSERT INTO TEMP SELECT FIRSTNAME...
  3. COMMIT

  이 기법을 통해 사용자는 기록하거나 카탈로그 경합을 벌이지 않고 대용량의 응답 세트를 테이블에 삽입할 수 있습니다. NOT LOGGED 상태를 활성화시키는 작업 단위에서 오류가 발생하면, 이것이 TEMP 테이블을 사용할 수 없도록 함에 주의하십시오. 이렇게 되면 TEMP 테이블을 제거하고 다시 작성해야 합니다. 이러한 이유로 사용자는 쉽게 재작성할 수 없는 테이블에 데이터를 추가하기 위해 이 기법을 사용해서는 *안됩니다*.

모든 노드로부터의 부분적 응답을 병합한 최종 응답 세트를 정렬할 필요가 있는 경우, 다음을 수행할 수 있습니다.

- 최종 SELECT에 SORT BY절을 지정하십시오.

- 각 파티션에서 별도의 파일로 추출을 수행하십시오.
- 예를 들어 sort -m AIX 명령을 사용하여 하나의 출력 세트로별도의 파일들을 병합하십시오.

테스트 환경 작성

DB2 Enterprise Edition을 사용하여 사용자의 파티션된 환경에 대한 테스트 환경을 작성할 수 있는데, 이는 DB2 Enterprise Edition이 DB2 Enterprise - Extended Edition과 같은 파티션 키 제한을 강제하기 때문입니다.

1. DB2 Enterprise Edition을 사용하여 사용자 데이터베이스 설계 모델을 작성하십시오.
2. 제품 환경의 파티션에 데이터를 분배하는 데 사용할 PARTITIONING KEY 절을 갖는 샘플 테이블을 작성하십시오.
3. 테스트 데이터베이스에 대하여 응용프로그램을 개발하고 수행하십시오.

DB2 Enterprise Edition은 DB2 Enterprise - Extended Edition과 일관된 파티션 키 제한조건을 강제하며, 사용자의 응용프로그램에 유용한 테스트 환경을 제공합니다.

오류 처리시 고려사항

파티션된 환경에서, DB2는 SQL문을 부속 절로 나누고, 이들 각각은 관련 데이터를 갖는 파티션에서 처리됩니다. 결과적으로 오류는 응용프로그램에 대한 액세스 권한이 없는 파티션에서 발생할 수 있습니다. 이것은 단일 파티션 환경에서는 발생하지 않습니다.

다음을 고려해야 합니다.

- 비 CURSOR (EXECUTE) 심각하지 않은 오류
- CURSOR 심각하지 않은 오류
- 심각한 오류
- 병합된 복수 SQLCA 구조
- 오류를 리턴하는 파티션 식별 방법

응용프로그램이 심각한 오류로 인해 이상 종료한 경우, 2단계 확약중 이상 실패 트랜잭션이 데이터베이스에 남아 있을 수 있습니다. 2단계 확약중 이상 실패 트랜잭션은 한 국면이 성공적으로 끝날 때 전역 트랜잭션과 관련되는데, 다음 단계가 완료되기 전에 시스템은 실패하고 데이터베이스는 일관성이 없는 상태로 남게 됩니다. 이것을 처리하는 것에 대한 정보는 *관리 안내서* 책을 참조하십시오.

심각한 오류

DB2 Universal Database에서 심각한 오류가 발생하면, 다음 중 하나가 발생합니다.

- 오류가 발생한 노드의 데이터베이스 관리 프로그램은 종료합니다.

  활동중인 작업 단위가 구간 복원되지 않습니다.

  이러한 상황에서 사용자는 종료가 발생했을 때 노드에서 활동중이던 노드나 데이터베이스를 복구해야 합니다.

- 오류가 발생한 노드의 데이터베이스에서 모든 에이전트들이 강제로 종료됩니다.

  이 데이터베이스의 모든 작업 단위는 구간 복원됩니다.

  이러한 경우, 오류가 발생한 노드의 데이터베이스는 일관성이 없는 것으로 표시됩니다. 이 데이터베이스에 액세스하려고 시도하면, SQLCODE -1034(SQLSTATE 58031) 또는 SQLCODE -1015(SQLSTATE 55025)가 리턴됩니다. 사용자나 다른 노드의 응용프로그램이 이 노드의 데이터베이스에 액세스할 수 있기 전에, 먼저 데이터베이스에 대하여 RESTART DATABASE 명령을 수행해야 합니다. 이 명령에 대한 정보는 *Command Reference* 책을 참조하십시오.

심각한 오류 SQLCODE -1224(SQLSTATE 55032)는 여러가지 이유로 발생할 수 있습니다. 이 메시지를 수신하는 경우, 어떤 노드가 실패했는지를 나타내는 SQLCA를 점검하십시오. 자세한 정보는 노드 간에 공유하는 `db2diag.log` 파일을 점검하십시오. 추가 정보는 639 페이지의 『오류를 리턴하는 파티션 식별』의 내용을 참조하십시오.

병합된 복수 SQLCA 구조

하나의 SQL문이 여러 노드에서 여러 에이전트에 의해 실행될 수 있으며, 각 에이전트는 다른 오류 또는 경고에 대하여 다른 SQLCA를 리턴할 수 있습니다. 조정

에이전트 역시 자신의 SQLCA를 갖습니다. 또한 SQLCA에는 행 갯수를 나타내는 *sqlerrd* 필드와 같은 전역 숫자를 나타내는 필드들이 있습니다. 응용프로그램에 일관성 있는 뷰를 제공하기 위해 모든 SQLCA 값은 하나의 구조로 병합됩니다. 이 구조는 *SQL 참조서*에서 설명됩니다.

오류 보고는 다음과 같습니다.

- 심각한 오류 조건은 항상 보고됩니다. 심각한 오류가 보고되면, 즉시 심각한 오류 이후 SQLCA에 추가가 수행되지 않습니다.
- 심각한 오류가 발생하지 않은 경우, 교착 상태 오류가 다른 오류에 비해 우선 순위가 높습니다.
- 기타 모든 다른 오류에 대하여, 첫번째 음수 SQLCODE에 대한 SQLCA가 응용프로그램에 리턴됩니다.
- 음수 SQLCODE가 탐지되지 않은 경우, 첫번째 경고(즉 양수 SQLCODE)에 대한 SQLCA가 응용프로그램에 리턴됩니다. 다른 파티션에는 데이터가 있으나 하나의 파티션에서 비어 있는 테이블에 대하여 데이터 조작 명령이 발행된 경우, 이에 대한 예외가 발생합니다. 모든 파티션에서의 에이전트가 SQL0100W를 리턴하는 경우 SQLCODE +100만 응용프로그램으로 리턴되는데, 이것은 모든 파티션에서 테이블이 비었거나 UPDATE문의 WHERE절을 충족시키는 행이 없기 때문입니다.
- 모든 오류와 경고에 대하여 *sqlwarn* 필드에는 모든 에이전트로부터 수신한 경고 플래그가 들어갑니다.
- 행 갯수를 나타내는 *sqlerrd* 필드의 값은 모든 에이전트의 누적 값입니다.

응용프로그램은 첫번째 오류나 경고를 발생시킨 문제점이 정정된 후 다시 그 뒤에 발생하는 오류나 경고를 수신할 수 있습니다. 처음 탐지된 오류에 다른 것보다 우선순위가 주어지도록 SQLCA에 오류가 보고됩니다. 즉 이전 오류에 의해 발생된 오류가 원래의 오류를 겹쳐쓰기 하지 않도록 합니다. 심각한 오류와 교착 상태는 조정 에이전트에 의한 즉각적인 조치를 필요로 하기 때문에 높은 우선순위가 주어집니다.

오류를 리턴하는 파티션 식별

파티션이 오류나 경고를 리턴하는 경우, 그 번호는 SQLCA의 SQLERRD(6) 필드에 들어가게 됩니다. 이 필드의 번호는 `db2nodes.cfg` 파일의 파티션에 지정된 것과 동일합니다.

SQL문이나 API가 성공적이라면 이 필드의 파티션 번호는 중요하지 않습니다.

SQLCA에 대한 정보는 *SQL 참조서* 책을 참조하십시오.

디버깅

응용프로그램 디버깅에는 다음에서 설명되는 도구를 사용할 수 있습니다. 자세한 정보는 *문제점 해결 안내서* 책을 참조하십시오.

루핑 또는 일시정지한 응용프로그램 진단

응용프로그램이나 조회를 시작한 후, 이 응용프로그램이 일시중지했거나(아무런 활동을 보이지 않는 경우), 루프를 돌고 있는(활동을 보이지만 응용프로그램으로 아무런 결과도 리턴되지 않는 경우) 것처럼 보일 수 있습니다. 잠금 시간종료를 켰는지 확인하십시오. 그러나 어떤 경우에는 오류가 리턴되지 않습니다. 이러한 상황에서 데이터베이스 시스템 모니터 스냅샷과 함께 *문제점 해결 안내서*에서 설명한 도구가 유용합니다.

응용프로그램 디버깅에 유용한 데이터베이스 시스템 모니터의 기능 중 하나는 모든 활동중인 에이전트의 상태를 표시하는 것입니다. 스냅샷을 가장 잘 활용하기 위해서는, 다음과 같이 응용프로그램을 수행하기 전에, 특히 DB2START를 수행한 직후에, 명령문 콜렉션이 수행되고 있는지 확인하십시오.

```
db2_all "db2 UPDATE MONITOR SWITCHES USING STATEMENT ON"
```

응용프로그램이나 조회가 정지했거나 루프를 돌고 있다고 판단되면, 다음의 명령을 발행하십시오.

```
db2_all "db2 GET SNAPSHOT FOR AGENTS ON database
```

스냅샷에서 수집된 정보를 읽는 방법 및 데이터베이스 시스템 모니터 사용에 대한 자세한 정보는 *시스템 모니터 안내 및 참조서* 책을 참조하십시오.

제19장 DB2 연합 시스템용 프로그램 작성

DB2 연합 시스템 소개 641
데이터 소스 테이블 및 뷰 액세스 642
  별명에 대한 작업 642
    데이터 소스 및 뷰에 대한 정보 카탈로그화 643
    고려사항 및 제한사항 643
    열 옵션 정의 645
    뷰에 별명 사용 646
  분리 레벨을 사용하여 데이터 무결성 유지보수 647
데이터 유형 맵핑에 대한 작업 647
  DB2에서 지역으로 정의할 데이터 유형을 판별하는 방법 648
  기본 데이터 유형 맵핑 648
  기본 유형 맵핑을 겹쳐쓰고 새로운 맵핑을 작성하는 방법 649
    하나 이상의 데이터 소스에 적용되는 유형 맵핑 정의 649
    특정 테이블에 대한 유형 맵핑 변경 649
분산 요청(DR)을 사용하여 데이터 소스 조회 650
  분산 요청(DR) 코딩 650
    부속 조회를 사용한 요청 651
    집합 연산자를 사용한 요청 651
    조인에 대한 요청 652
  서버 옵션을 사용하여 최적화 촉진 . . . 652
데이터 소스 함수 호출 653
  데이터 소스 함수를 호출하도록 DB2 설정 653
  함수 호출의 오버헤드 감소. 654
  CREATE FUNCTION MAPPING문에서 함수 이름 지정 656
  함수 맵핑 분리 656
Pass-Through를 사용하여 데이터 소스 직접 조회 656
  Pass-Through 세션으로 SQL 처리 . . . 656
  고려사항 및 제한사항 657
    모든 데이터 소스에 Pass-Through 사용 657
    Oracle 데이터 소스에 Pass-Through 사용 658

DB2 연합 시스템 소개

*DB2 연합 시스템*은 다음으로 구성된 분산 컴퓨팅 시스템입니다.

- *연합 서버*라 하는 DB2 서버
- 연합 서버에서 조회를 전송하는 복수 반자율 데이터 소스. 각 데이터 소스는 관계형 데이터베이스 관리 시스템의 인스턴스과 인스턴스에서 지원하는 데이터베이스(들)로 구성됩니다. DB2 연합 시스템의 데이터 소스에는 Oracle 인스턴스와 DB2 계열 구성원의 인스턴스가 들어갈 수 있습니다.

클라이언트 응용프로그램에 데이터 소스는 단일 집합 데이터베이스로 나타납니다. 그러나 응용프로그램은 연합 서버에 있는 *연합 데이터베이스*라 하는 데이터베이스와 실제로 상호작용합니다. 데이터 소스의 데이터를 얻으려면 연합 데이터베이스

© Copyright IBM Corp. 1993, 2000

에 조회를 DB2 SQL로 제출하십시오. 그러면 DB2에서 해당 데이터 소스로 조회를 분배하고, 요청된 데이터를 수집하고, 이 데이터를 응용프로그램으로 리턴합니다.

응용프로그램에서 DB2 SQL을 사용하여 DB2에서 인식할 수 있는 데이터 유형(LOB 데이터 유형 제외)의 값을 요청할 수 있습니다. 데이터 소스 테이블을 갱신할 경우와 같이 데이터 소스에 기록하려면 응용프로그램에서 데이터 소스의 SQL을 pass-through라 하는 특수 모드로 사용해야 합니다.

연합 데이터베이스의 시스템 카탈로그에는 데이터베이스의 오브젝트에 대한 정보뿐 아니라 데이터 소스와 특정 테이블, 뷰 및 그 기능에 대한 정보도 있습니다. 카탈로그에는 전체 연합 시스템에 대한 정보가 있으므로 *전역 카탈로그*라 합니다.

DB2 연합 시스템에 대한 상세한 개요는 *관리 안내서: 계획*에서 자세한 내용을 참조하십시오. 확장 개요에 대해서는 *SQL 참조서*에서 자세한 내용을 참조하십시오. 응용프로그램에서 제출할 수 있는 DB2 SQL 조회에 대해서는 650 페이지의 『분산 요청(DR)을 사용하여 데이터 소스 조회』에서 자세한 내용을 참조하십시오. Pass-Through에 대해서는 656 페이지의 『Pass-Through를 사용하여 데이터 소스 직접 조회』에서 자세한 내용을 참조하십시오.

데이터 소스 테이블 및 뷰 액세스

이 절에는 데이터 소스 테이블과 뷰에 액세스하여 사용하는 데 유용한 정보가 있습니다. 이 절에서 다루는 주제는 다음과 같습니다.

- 연합 서버에서 참조할 수 있도록 테이블과 뷰에 할당하는 별명
- 테이블과 뷰에 액세스할 때 데이터 소스에서 데이터 무결성을 유지보수하는 데 유용한 분리 레벨

별명에 대한 작업

*별명*은 응용프로그램에서 데이터 소스 테이블이나 뷰를 참조하는 데 사용할 수 있는 식별자입니다. 이 절에서는

- 별명을 작성할 테이블에 대한 정보를 전역 카탈로그에 제공하는 방법을 설명합니다.

- 별명에 대해 작업할 때 기억해야 할 고려사항과 제한사항을 나열합니다.
- 조회를 최적화할 때 설정할 수 있는 매개변수에 대해 설명합니다.
- 별명으로 참조하는 뷰 사용 방법에 대해 설명합니다.

데이터 소스 및 뷰에 대한 정보 카탈로그화

데이터 소스나 뷰에 대해 별명을 작성하면 DB2에서 테이블이나 뷰의 데이터를 검색하는 방법을 계획할 때 최적화 알고리즘이 사용할 수 있는 정보로 전역 카탈로그를 갱신합니다. 이 정보에는 테이블이나 뷰의 이름, 테이블 또는 뷰 컬럼의 이름과 속성 등이 포함됩니다.

테이블의 경우 이 정보에는 다음도 포함됩니다.

- 행이 있는 페이지의 수나 행 수와 같은 통계. DB2에서 최신 통계를 확보하게 하려면 별명을 작성하기 전에 테이블에 대해 데이터 소스의 RUNSTATS 명령 해당 부분을 수행하는 것이 좋습니다.
- 테이블의 색인에 대한 설명. 테이블에 색인이 없는 경우에도, 고유 값을 가지는 테이블의 컬럼(들)이나 행이 고유한 지 여부와 같이 색인 정보에 포함되는 메타데이터를 카탈로그에 제공할 수 있습니다. 테이블의 별명에 대해 CREATE INDEX문을 수행하여, 집합적으로 *색인 스펙*이라 하는 이 메타데이터를 생성할 수 있습니다. 명령문을 사용하면 색인 스펙만 생성되고 실제 색인은 작성되지 않습니다. 이 점에 주의하십시오. 이 명령문에 대해서는 *SQL 참조서*에서 자세한 내용을 참조하십시오.

전역 카탈로그에 저장된 데이터 소스 테이블에 대한 정보를 알아내려면 SYSCAT.TABLES와 SYSCAT.COLUMNS 카탈로그 뷰를 조회하십시오. 카탈로그에 저장된 테이블 색인에 대한 정보나 특정 색인 스펙에 포함된 내용을 알아내려면 SYSCAT.INDEXES 카탈로그 뷰를 조회하십시오. 이런 뷰에 대해서는 *SQL 참조서*에서 자세한 내용을 참조하십시오. 테이블 및 색인에 대한 정보를 사용한 전역 카탈로그 갱신에 대해서는 *관리 안내서: 구현*의 내용을 참조하십시오.

고려사항 및 제한사항

다음을 실행할 때 염두에 두어야 할 고려사항과 제한사항이 몇 가지 있습니다.

- 별명 정의, 변경 및 제거
- 테이블과 뷰를 별명으로 참조

- 별명으로 참조한 테이블과 뷰에 대한 작업 수행

별명 정의, 변경 및 제거:

- 테이블이나 뷰의 별명을 정의하려면 CREATE NICKNAME문을 참조하십시오. 이 명령문에서
  - 그 이름으로 Oracle 테이블이나 뷰를 참조합니다.
  - 그 이름으로 또는 별명이 있을 경우 그 별명으로 DB2 계열 테이블이나 뷰를 참조할 수 있습니다.
- 동일 테이블이나 뷰에 대해 별명을 하나 이상 정의할 수 있습니다. 또한 CREATE ALIAS문으로 별명의 별명을 정의할 수도 있습니다.
- 별명을 변경하려면 별명을 제거한 다음 대체해야 합니다. 별명을 제거하려면 DROP NICKNAME문을 사용하고 별명을 대체하려면 CREATE NICKNAME 문을 사용하십시오.
- 별명을 제거하면 별명을 사용하여 정의한 모든 뷰가 작동 불능 상태에 빠져 그 뷰에 종속된 모든 플랜은 유효하지 않게 됩니다.

CREATE NICKNAME, CREATE ALIAS 및 DROP NICKNAME문에 대해서는 *SQL 참조서*에서 자세한 내용을 참조하십시오.

별명으로 테이블과 뷰 참조:

- 데이터 소스 테이블이나 뷰에 별명을 지정하면 그 별명만으로 테이블이나 뷰를 참조할 수 있습니다(pass-through 세션 제외). 예를 들어, DB2MVS1.PERSON. DEPT 테이블을 표시하도록 DEPT 별명을 정의할 경우 SELECT * FROM DEPT문이 허용되지만 SELECT * FROM DB2MVS1.PERSON.DEPT는 허용되지 않습니다. 그러나 pass-through 세션에서는 데이터 소스 이름으로 테이블이나 뷰에 액세스해야 합니다.
- CREATE TRIGGER문의 별명을 참조할 수 없습니다.
- 요약 테이블 정의 절에서 별명을 참조할 경우 이 절에서 DEFINITION ONLY 키워드도 지정해야 합니다.

별명이 있는 테이블 및 뷰에 대한 작업 수행:

- COMMENT ON문이 별명 및 별명에 대해 정의된 열에 유효합니다. 이 명령문은 전역 카탈로그를 갱신하지만 데이터 소스 카탈로그는 갱신하지 않습니다.

- GRANT 및 REVOKE문은 특정 특권 및 모든 사용자와 그룹에 대해 별명에 유효합니다. 그러나 DB2에서는 별명이 참조하는 테이블이나 뷰에 대해 해당 GRANT 또는 REVOKE를 발행하지 않습니다. 별명 특권에 대해서는 *관리 안내서: 계획*에서 자세한 내용을 참조하십시오.
- 데이터 소스는 읽기 전용입니다. 따라서
  - INSERT, UPDATE 및 DELETE문은 별명에 대해 유효하지 않습니다.
  - 별명에 대한 UNION ALL절이 있는 뷰는 갱신할 수 없습니다.
- RUNSTATS, IMPORT, EXPORT 등과 같은 DB2 유틸리티를 별명에 대해 사용할 수 없습니다.

열 옵션 정의

테이블이나 뷰의 별명을 정의하면 테이블이나 뷰의 특정 컬럼에 대한 정보를 전역 카탈로그에 제공할 수 있습니다. 이 정보는 *컬럼 옵션* 매개변수에 할당하는 값의 양식으로 지정합니다. 이런 값은 대문자나 소문자로 지정할 수 있습니다. 표27에서는 컬럼 옵션과 그 값에 대해 설명합니다.

표 27. 컬럼 옵션과 설정

옵션	유효한 설정	기본 설정
numeric_string	‘y’ 예, 이 컬럼에는 숫자 데이터의 문자열만 있습니다. 중요: 이 컬럼에 숫자 문자열만 있고 그 뒤에 뒤 공백이 나올 경우 ‘y’를 지정하지 않는 것이 좋습니다. ‘n’ 아니오, 이 컬럼은 숫자 데이터의 문자열에 제한되지 않습니다. 컬럼에 대해 numeric_string을 ‘y’로 설정하여 이 컬럼에 공백이 없어 컬럼 데이터의 정렬에 대해 상호작용할 수 있다는 사실을 최적화 알고리즘에 알려 줄 수 있습니다.	‘n’

표 27. 컬럼 옵션과 설정 (계속)

옵션	유효한 설정	기본 설정
varchar_no_trailing_blanks	특정 VARCHAR 컬럼부터 뒤 공백이 없는지 여부를 지시합니다. 'y' 예, 이 VARCHAR 컬럼부터 뒤 공백이 없습니다. 'n' 아니오, 이 VARCHAR 컬럼부터 뒤 공백이 없습니다. 데이터 소스 VARCHAR 컬럼에 공간을 채우는 공백이 없을 경우 그에 액세스하는 최적화 알고리즘의 전략은 부분적으로 뒤 공백이 있는지 여부에 의존합니다. 기본적으로 최적화 알고리즘에서는 실제로 뒤 공백이 있다고 『가정합니다』. 이런 가정에서 최적화 알고리즘은 이 컬럼의 리턴 값이 사용자가 예상한 값이 되도록 수정 조회를 포함하는 액세스 전략을 개발합니다. 그러나 VARCHAR 컬럼에 뒤 공백이 없으며 최적화 알고리즘에 이 사실을 인식시켜 주면 최적화 알고리즘에서 더 효율적인 액세스 전략을 개발할 수 있습니다. 특정 컬럼에 뒤 공백이 없다는 사실을 최적화 알고리즘에 알려 주려면 ALTER NICKNAME문에서 그 컬럼을 지정하십시오(지침에 대해서는 *SQL 참조서* 참조).	'n'

ALTER NICKNAME문에서 컬럼 옵션을 설정합니다. 이 명령문에 대해서는 *SQL 참조서*에서 자세한 내용을 참조하십시오.

뷰에 별명 사용

두 가지 방법으로 뷰에 별명을 사용할 수 있습니다.

- 데이터 소스 뷰의 별명을 작성할 수 있습니다. 연합 서버에서는 데이터 소스 테이블의 별명을 다룰 때와 동일한 방식으로 데이터 소스 뷰의 별명을 다룹니다.
- 별명을 가진 데이터 소스 테이블과 뷰의 연합 데이터베이스 뷰를 작성할 수 있습니다. 예를 들어, 연합 서버가 다른 위치에서 기본 테이블의 조인을 수용할 수 있으므로 다른 데이터 소스에 있는 기본 테이블의 연합 데이터베이스 뷰를 쉽게 정의할 수 있습니다. 이런 복수 위치 뷰에서는 복수 지역 테이블에 정의된 뷰가 중앙 관계형 데이터베이스 관리 프로그램에 대해 실행할 때와 마찬가지로 전역으로 통합된 데이터베이스에 대해 고도의 데이터 독립성을 제공합니다. 이 전역 뷰 메카니즘은 연합 서버에서 고도의 데이터 독립성을 제공하는 방식입니다.

데이터 소스 테이블에 대한 연합 데이터베이스 뷰를 작성하는 조치를 때로 『별명에 대한 뷰 작성』이라 합니다. 즉, 뷰를 작성하려면 CREATE VIEW문의 fullselect에서 뷰에 들어갈 각 테이블과 뷰의 별명을 참조해야 합니다.

데이터베이스에 있는 실제 테이블이 아닐 경우 뷰에는 자체 통계나 색인이 없습니다. 구조와 내용면에서 뷰가 단일 기본 테이블과 같을 경우에도 이 명령문은 해당됩니다. 통계와 색인에 대해서는 *관리 안내서: 구현*의 내용을 참조하십시오.

분리 레벨을 사용하여 데이터 무결성 유지보수

특정 분리 레벨에서 테이블의 행 잠금을 요청하여 데이터 소스 테이블에 대한 데이터 무결성을 유지보수할 수 있습니다. 예를 들어, 행에 대한 단일 액세스가 있는지 확인하려면 그 행에 대한 반복 읽기(RR) 분리 레벨을 지정해야 합니다.

연합 서버에서 사용자가 요청한 분리 레벨을 데이터 소스의 해당 부분에 맵핑합니다. 이를 나타내기 위해 표28에서 다음을 나열합니다.

- 사용자가 요청할 수 있는 분리 레벨은 다음과 같습니다.

  CS　　커서 안정성

  RR　　반복 읽기

  RS　　읽기 안정성

  UR　　미확약 읽기

- 요청된 레벨이 맵핑되는 Oracle 분리 레벨.

표 28. 연합 서버와 Oracle 데이터 소스 사이의 비교 가능한 분리 레벨

연합 서버 (DB2)	CS	RR	RS	UR
Oracle	기본값	트랜잭션 읽기 전용	트랜잭션 읽기 전용	커서 안정성 (CS)과 동일

데이터 유형 맵핑에 대한 작업

데이터 소스 테이블의 별명을 작성할 때 DB2에서 테이블에 대한 정보로 전역 카탈로그를 이식합니다. 이런 정보에는 별명, 테이블의 이름, 모든 컬럼 이름 등이 포함되며 각 컬럼에 대해 다음의 내용이 포함됩니다.

- 데이터 소스에서 컬럼에 대해 정의된 데이터 유형(이 절에서는 이 유형을 *원격 유형*이라 합니다.)
- DB2에서 지원하며 연합 데이터베이스에 등록된 해당 데이터 유형(이 절에서는 이 유형을 *지역* 유형이라 합니다.)

이 절에서는 DB2에서 데이터 유형 맵핑을 사용하여 데이터 소스 테이블의 컬럼에 대해 정의된 DB2 지원 데이터 유형을 판별하는 방법에 대해 설명합니다. 그런 다음 두 하위 절에서 데이터 유형 맵핑(『유형 맵핑』이라고 함)에 대해 논의합니다. 첫번째 하위 절에서는 기본 맵핑에 대해 설명하고, 두 번째 절에서는 기본 맵핑을 겹쳐쓰고 새로운 맵핑을 작성하는 방법을 나타냅니다.

DB2에서 지역으로 정의할 데이터 유형을 판별하는 방법

DB2에서 원격 컬럼에 대해 사용할 지역 유형을 판별하는 방법은? 이 문장은 데이터 소스의 컬럼 유형과 비교 가능한 지역 유형 사이의 맵핑을 의미하며 후자를 선택합니다. 예를 들어, DB2에서 제공하는 기본 맵핑에서, 254 바이트까지 지원하는 CHAR VSE & VM용 DB2 데이터 유형은 DB2 CHAR 데이터 유형을 지시합니다. 따라서 VSE & VM용 DB2 테이블의 별명을 작성했는데 테이블의 C1 컬럼에 최대 길이가 200인 CHAR 데이터 유형이 있을 경우 기본값을 겹쳐쓰지 않는 한 C1에 대해 CHAR DB2 유형이 지역으로 정의됩니다.

기본 데이터 유형 맵핑

RDBMS 사이의 차이점 때문에 데이터 소스 데이터 유형과 연합 서버 데이터 유형 사이의 기본 맵핑이 항상 1 대 1은 아닙니다. 그러나 요청된 모든 값이 리턴되도록 보장할만큼 맵핑은 근접합니다.

예를 들어, 다음 사이에 기본 유형 맵핑이 있습니다.

- Oracle 유형 NUMBER(9,0)(여기서 9는 최대 정밀도이고 0은 최대 스케일)
- DB2 유형 INTEGER, 최대 길이는 4 바이트

NUMBER(9,0) 유형을 포함한 열이 있는 Oracle 테이블의 별명을 작성한다고 가정해 보십시오. 기본 맵핑을 변경하지 않을 경우 C2의 유형은 지역으로 INTEGER로 정의됩니다. 또한 INTEGER의 4바이트가 최대 정밀도 10을 지원하므로, 연합 서버에서 C2를 조회할 때 C2의 모든 값이 리턴된다고 확신할 수 있습니다.

기본 데이터 유형의 목록에 대해서는 *SQL 참조서*에서 자세한 내용을 참조하십시오.

기본 유형 맵핑을 겹쳐쓰고 새로운 맵핑을 작성하는 방법

앞의 예에서 지적했듯이, 정의된 원격 유형에 대해 원격 컬럼을 조회할 때 두 유형을 따르는 모든 값이 리턴될 정도로 기본 맵핑의 지역 유형과 원격 유형은 비슷합니다. 그러나 때로 대체 맵핑을 요구할 수 있습니다. 다음 시나리오를 참조하십시오.

하나 이상의 데이터 소스에 적용되는 유형 맵핑 정의

Oracle 데이터 소스의 세 테이블에 있는 특정 컬럼에는 시간소인에 대한 DATE 데이터 유형이 있습니다. 기본 맵핑에서 이 유형은 지역 DB2 유형 TIMESTAMP를 지시합니다. 따라서 기본값을 변경하지 않고 세 테이블의 별명을 작성하려는 경우 이런 컬럼에 대해 TIMESTAMP가 지역으로 정의되고 컬럼에 대한 DB2 조회가 시간소인을 생성합니다. 그러나 이런 조회로 시간만 생성하려 한다고 가정해봅시다., 이 경우, 기본값을 겹쳐쓰면서 Oracle DATE를 DB2 유형 TIME에 맵핑할 수 있습니다. 이런 방식으로 별명을 작성하면 컬럼에 대해 TIMESTAMP가 아닌 TIME이 지역으로 정의됩니다. 따라서 이를 조회하면 시간소인의 시간 부분만 리턴됩니다. 기본 유형 맵핑을 겹쳐쓰려면 CREATE TYPE MAPPING문을 사용해야 합니다.

CREATE TYPE MAPPING문에서, 원하는 새로운 맵핑을 사용자의 회사 부서에서 사용하는 데이터 소스와 같은 특정 데이터 소스에 적용할 지 아니면 모든 Oracle 데이터 소스와 같은 특정 유형의 모든 데이터 소스에 적용할 지 아니면 모든 Oracle 8.0.3 데이터 소스와 같이 특정 유형 버전의 모든 데이터 소스에 적용할 지 지정할 수 있습니다.

특정 테이블에 대한 유형 맵핑 변경

특정 테이블에 대한 유형 맵핑에서 지역 유형을 변경할 수 있습니다. 예를 들어, Oracle 데이터 유형 NUMBER(32,3)에서는 기본적으로 DB2 데이터 유형 DOUBLE, 부동 소수 데이터 유형에 맵핑됩니다. 직원 정보에 대한 Oracle 테이블에서 NUMBER(32,3) 데이터 유형으로 BONUS 컬럼을 정의했다고 가정해봅시다. 맵핑 때문에 BONUS에 대한 조회는 다음과 같은 값을 리턴할 수 있습니다.

```
5.0000000000000E+002
1.0000000000000E+003
```

여기서 +002는 소수점을 오른쪽으로 두 자리 이동해야 한다는 사실을 지시하고, +003은 소수점을 오른쪽으로 세 자리 이동해야 한다는 사실을 지시합니다.

따라서 BONUS에 대한 조회로 달러 금액과 같은 값을 리턴할 수 있으므로 이 특정 테이블에 대해 실제 보너스의 형식을 반영하는 정밀도와 스케일을 포함한 DB2 DECIMAL 유형에 NUMBER(32,3)를 다시 맵핑할 수 있습니다. 예를 들어, 보너스의 달러 부분이 6자리를 넘지 않는다는 사실을 알 경우 DECIMAL(8,2)에 NUMBER(32,3)를 다시 맵핑할 수 있습니다. 이 새로운 맵핑의 제한조건 아래에서 BONUS에 대한 조회는 다음과 같은 값을 리턴합니다.

```
 500.00
1000.00
```

특정 테이블의 컬럼에 대한 유형 맵핑을 변경하려면 AFTER NICKNAME문을 사용하십시오. 이 명령문을 사용하여 별명을 정의한 테이블의 컬럼에 대해 지역으로 정의한 유형을 변경할 수 있습니다.

분산 요청(DR)을 사용하여 데이터 소스 조회

연합 데이터베이스에 제출된 조회로 단일 데이터 소스로 생성되는 결과를 요청할 수 있지만 보통 복수 데이터 소스로 생성되는 결과를 요청합니다. 일반적인 조회는 복수 데이터 소스에 분산되므로 *분산 요청*이라 합니다.

이 절에서는

- 분산 요청을 코딩하는 방법을 나타냅니다.
- 특정 분산 요청의 최적화를 돕는 방법을 소개합니다.

분산 요청(DR) 코딩

일반적으로 분산 요청은 세 가지 SQL 표기법 중 하나 이상을 사용하여 부속 조회, 집합 연산자 및 조인 subselect 등 데이터를 검색할 위치를 지정합니다. 이 절에는 다음 시나리오 문맥의 예가 있습니다. 연합 서버는 OS/390용 DB2 Universal Database 데이터 소스, AS/400용 DB2 Universal Database 데이터 소스 및 Oracle 데이터 소스에 액세스하도록 구성되어 있습니다. 직원 정보를 포함한 테이블은 각

데이터 소스에 저장됩니다. 연합 서버에서는 테이블이 있는 위치를 지시하는 별명 UDB390_EMPLOYEES, AS400_EMPLOYEES 및 ORA_EMPLOYEES로 이런 테이블을 참조합니다.(별명은 데이터 소스를 참조할 필요가 없습니다. 이 시나리오의 별명은 테이블이 다른 RDBMS에 있다는 사실만을 강조하기 위해 참조합니다.) ORA_EMPLOYEES 외에, Oracle 데이터 소스에는 직원이 사는 국가에 대한 정보를 포함한 ORA_COUNTRIES 별명의 테이블이 있습니다.

부속 조회를 사용한 요청

AS400_EMPLOYEES 테이블에는 아시아에 사는 직원의 전화 번호가 있습니다. 또한 이런 전화 번호와 연관된 국가 코드도 있지만 코드가 표시하는 테이블에서 국가는 나열하지 않습니다. 코드와 국가는 ORA_COUNTRIES 테이블에 나열되어 있습니다. 다음 조회에서는 부속 조회를 사용하여 중국의 국가 코드를 알아내며 SELECT와 WHERE절을 사용하여, 다음과 같은 특정 코드가 필요한 전화 번호를 가진 AS400_EMPLOYEES의 직원을 나열합니다.

```
SELECT name, telephone
   FROM djadmin.as400_employees
   WHERE country_code IN
      (SELECT country_code
         FROM djadmin.ora_countries
         WHERE country_name = 'CHINA')
```

집합 연산자를 사용한 요청

연합 서버에서는 다음과 같이 세 가지 집합 연산자를 지원합니다.

- UNION

  이 집합 연산자를 사용하면 SELECT문을 두 개 이상 만족시키는 행을 결합할 수 있습니다.

- EXCEPT

  이 집합 연산자를 사용하면 두 번째가 아닌 첫번째 SELECT문을 충족시키는 행을 검색할 수 있습니다.

- INTERSECT

  이 집합 연산자를 사용하면 두 SELECT문을 충족시키는 행을 검색할 수 있습니다.

중복 행이 결과에서 제거되지 않아 추가 정렬할 필요가 없음을 지시하는 ALL 피연산자가 세 집합 연산자 모두에 포함될 수 있습니다.

각 테이블이 다른 데이터 소스에 있는 경우에도 다음 조회는 AS400_EMPLOYEES 및 UDB390_EMPLOYEES에 나타난 모든 직원 이름과 국가 코드를 검색합니다.

```
SELECT name, country_code
   FROM as400_employees
INTERSECT
SELECT name, country_code
   FROM udb390_employees
```

조인에 대한 요청

관계형 조인은 두 개 이상의 테이블에서 검색한 컬럼의 조합이 있는 결과 세트를 생성합니다. 결과 세트 행의 크기를 제한하는 조건을 지정해야 합니다.

아래 조회는 두 테이블에 나열된 국가 코드를 비교하여 직원 이름과 해당 국가 이름을 결합합니다. 각 테이블은 다른 데이터 소스에 있습니다.

```
SELECT t1.name, t2.country_name
   FROM djadmin.as400_employees t1, djadmin.ora_countries t2
   WHERE t1.country_code = t2.country_code
```

서버 옵션을 사용하여 최적화 촉진

연합 시스템 사용자는 *서버 옵션*이라 하는 매개변수를 사용하여 데이터 소스에 전체로 적용되는 정보를 전역 카탈로그에 제공하거나 DB2와 데이터 소스 사이의 상호작용을 제어할 수 있습니다. 예를 들어, 데이터 소스의 기본 역할을 하는 인스턴스의 식별자를 카탈로그화하기 위해 데이터베이스 관리자가 서버 옵션 『노드』에 값으로 식별자를 할당합니다.

몇 가지 서버 옵션은 DB2와 데이터 소스 사이의 주요 영역, 조회 최적화를 나타냅니다. 예를 들어, 『varchar_no_trailing_blanks』 컬럼 옵션을 사용하여 DB2 최적화 알고리즘에 뒤 공백이 없는 특정 데이터 소스 VARCHAR 컬럼을 알려 줄 수 있듯이, 『varchar_no_trailing_blanks』라고도 하는 서버 옵션을 사용하여 데이터 소스의 VARCHAR 컬럼에 뒤 공백이 없다는 사실을 최적화 알고리즘에 알려 줄 수 있습니다. 이런 정보를 사용하여 최적화 알고리즘으로 액세스 전략을 작성하는 방법에 대해서는 645 페이지의 표27에서 자세한 내용을 참조하십시오.

또한 Oracle 데이터 소스에 *플랜 힌트*라 하는 명령문 일부를 제공하도록 DB2를 설정하는 값으로 서버 옵션 『plan_hints』를 설정할 수 있습니다. 플랜 힌트는 Oracle 최적화 알고리즘의 작업을 돕습니다. 특히 플랜 힌트를 통해 최적화 알고리즘은 테

이블에 액세스할 때 사용할 색인이나 결과 세트의 데이터를 검색할 때 사용할 테이블 조인 순서와 같은 문제를 쉽게 결정할 수 있습니다.

일반적으로 데이터베이스 관리자가 연합 시스템에 대해 서버 옵션을 설정합니다. 그러나 프로그래머가 조회를 최적화하는 데 유용한 옵션을 활용할 수도 있습니다. 예를 들어, ORACLE1과 ORACLE2 데이터 소스의 경우 plan_hints 서버 옵션은 기본값 'n'(아니오, 이 데이터 소스에 플랜 힌트를 제공하지 않습니다)으로 설정됩니다. 또한 ORACLE1과 ORACLE2 데이터에 대한 분산 요청을 작성하며, 이런 데이터 소스의 최적화 알고리즘에서 이 데이터에 액세스하는 전략을 향상시키는 데 플랜 힌트가 유용하리라 예상할 경우를 가정해봅시다. 응용프로그램이 연합 데이터베이스에 연결되어 있는 동안 'y'(예, 플랜 힌트를 제공합니다)의 설정으로 기본값을 겹쳐쓸 수 있습니다. 연결이 완료되면 설정이 자동으로 'n'으로 반전됩니다.

연합 데이터베이스에 대한 연결 중 서버 옵션 설정을 강제하려면 SET SERVER OPTION문을 사용하십시오. 설정을 적용하려면 CONNECT문 뒤 오른쪽에 명령문을 지정해야 합니다. 또한 명령문을 동적으로 준비하는 것이 좋습니다.

SET SERVER OPTION문에 대해서는 *SQL 참조서*에서 자세한 내용을 참조하십시오. 모든 서버 옵션과 그 설정에 대해서는 *관리 안내서: 구현*의 내용을 참조하십시오.

데이터 소스 함수 호출

이 절에서는 다음 방법을 설명합니다.

- 인식하지 못하는 데이터 소스 함수를 호출하도록 DB2 설정
- DB2에서 함수를 호출할 때 소비되는 오버헤드의 감소
- 서로 맵핑하려는 함수의 이름 지정
- 함수 사이의 맵핑 사용 중단

데이터 소스 함수를 호출하도록 DB2 설정

때로 DB2에서 인식하지 못하는 데이터 소스 함수를 DB2로 호출하려 할 수 있습니다. 이런 함수는 DB2에 알려지지 않는 새로 내장한 함수나 사용자 정의 함수일 수 있습니다.

DB2에서 인식하지 못하는 데이터 소스 함수에 액세스하려면 먼저 이 함수와 연합 데이터베이스에 저장된 함수 사이의 맵핑을 작성해야 합니다. 맵핑을 작성하려면 상대 함수를 선택한 다음 맵핑을 작성할 DDL문을 제출하십시오. 이 명령문을 CREATE FUNCTION MAPPING이라 합니다.

상대 함수는 기존 함수나 함수 템플리트나 사용자가 작성한 함수 또는 함수 템플리트일 수 있습니다.(*함수 템플리트*는 실행 코드가 없는 부분적인 함수입니다.) CREATE FUNCTION문으로 함수나 함수 템플리트를 작성할 수 있습니다.

데이터 소스 함수와 연합 데이터베이스 상대 함수가 다음 방식으로 통신해야 합니다.

- 두 함수의 입력 매개변수 수가 같아야 합니다.
- 데이터 소스 함수의 입력 매개변수의 데이터 유형은 연합 데이터베이스 상대 함수의 데이터 유형과 호환 가능해야 합니다.

CREATE FUNCTION MAPPING 및 CREATE FUNCTION문에 대해서는 *SQL 참조서*에서 자세한 내용을 참조하십시오.

함수 호출의 오버헤드 감소

데이터 소스 함수에 연합 서버 함수를 맵핑하려면—CREATE FUNCTION MAPPING문—DDL에서 데이터 소스 함수를 호출할 때 소비되는 오버헤드에 대한 추정 통계를 포함할 수 있어야 합니다. 예를 들어, 명령문에서 데이터 소스를 호출하는 데 필요한 지침의 추정치와 이 함수로 전달되는 인수 집합의 각 바이트에 대해 소비되는 I/O의 추정치를 지정할 수 있습니다. 이런 추정치는 전역 카탈로그에 저장되며 SYSCAT.FUNCMAPOPTIONS 뷰에서 볼 수 있습니다. 또한 DB2 함수(템플리트가 아닌)에서 맵핑에 참여할 경우 이 함수를 호출할 때 소비되는 오버헤드의 추정치가 카탈로그에 포함됩니다. 후자 추정치는 SYSCAT. FUNCTIONS 뷰에서 볼 수 있습니다.

맵핑을 작성하면 DB2 함수를 참조하는 분산 요청을 제출할 수 있습니다. 예를 들어, DOLLAR라는 DB2 사용자 정의 함수를 US_DOLLAR라 하는 Oracle 사용자 정의 함수에 맵핑할 경우 요청에서 US_DOLLAR가 아닌 DOLLAR를 지정합니다. 요청이 처리되면 최적화 알고리즘에서 복수 액세스 전략을 평가합니다. 그

중 일부는 DB2 함수를 호출하는 추정 오버헤드를 반영합니다. 그 외의 함수는 데이터 소스 함수를 호출하는 추정 오버헤드를 반영합니다. 최소 오버헤드를 지출하는 것으로 간주되는 전략이 사용됩니다.

소비되는 오버헤드의 추정치가 변경되면 변경사항을 전역 카탈로그에 기록할 수 있습니다. 데이터 소스 함수의 새로운 추정치를 기록하려면 먼저 함수 맵핑을 제거하거나 사용안함으로 설정하십시오(그 방법에 대해서는 656 페이지의 『함수 맵핑 분리』 참조). 그런 다음 CREATE FUNCTION MAPPING문에서 새로운 추정치를 지정하면서 그 명령문으로 맵핑을 다시 작성하십시오. 명령문을 실행하면 새로운 추정치가 SYSCAT.FUNCTIONS 카탈로그 뷰에 추가됩니다. DB2 기능의 변경된 추정치를 기록하려면 SYSSTAT.FUNCTIONS 카탈로그 뷰를 직접 갱신하십시오.

CREATE FUNCTION MAPPING문의 추정된 통계를 *함수 맵핑 옵션*이라는 매개변수의 값으로 할당하여 추정된 통계를 지정합니다. 표29에서 이런 옵션과 그 값에 대해 설명합니다.

표 29. 함수 맵핑 옵션과 설정

옵션	유효한 설정	기본 설정
ios_per_invoc	데이터 소스 함수 호출 당 I/O의 추정치	‘0’
insts_per_invoc	데이터 소스 함수 호출 당 처리되는 지침의 추정치	‘450’
ios_per_argbyte	데이터 소스 함수에 전달된 인수 집합의 각 바이트에 대해 소비되는 I/O의 추정치	‘0’
insts_per_argbyte	데이터 소스 함수에 전달된 인수 집합의 각 바이트에 대해 처리되는 지침의 추정치	‘0’
percent_argbytes	데이터 소스 함수에서 실제로 읽는 입력 인수 바이트의 평균 추정 백분율	‘100’
initial_ios	데이터 소스 함수를 호출했을 때 처음과 마지막으로 수행되는 I/O의 추정치	‘0’
initial_insts	데이터 소스 함수를 호출했을 때 처음과 마지막으로 처리되는 지침의 추정치	‘0’

DROP FUNCTION MAPPING문, SYSCAT.FUNCTIONS와 SYSSTAT.FUNCTIONS 뷰 및 SYSCAT.FUNCMAPOPTIONS 뷰에 대해서는 *SQL 참조서*에서 자세한 내용을 참조하십시오.

CREATE FUNCTION MAPPING문에서 함수 이름 지정

CREATE FUNCTION MAPPING문을 코딩하는 방법은 부분적으로 함께 맵핑하는 오브젝트의 이름이 같은 지 다른 지 여부에 따라 달라집니다. 같은 이름을 가진 두 함수(또는 함수 템플리트와 함수) 사이의 맵핑을 작성할 경우 *function-name* 매개변수에 이 이름을 할당해야 합니다.

이름이 다를 경우

- *function-name* 매개변수에 연합 데이터베이스 함수나 함수 템플리트의 이름을 할당하십시오.
- 『remote_name』이라는 함수 맵핑 옵션을 지정하고 이 옵션에 데이터 소스 함수의 이름을 할당하십시오. 이름은 225자보다 적어야 합니다.

함수 맵핑 분리

함수 맵핑을 사용하여 분리하려면 다음 지침을 따르십시오.

- 맵핑이 SYSCAT.FUNCMAPPINGS 카탈로그 뷰에 나열된 경우 맵핑을 삭제하십시오. DROP FUNCTION MAPPING문으로 삭제할 수 있습니다.
- SYSCAT.FUNCMAPPINGS 뷰에 나열되어 있지 않은 기본 맵핑을 분리하려면 맵핑을 사용안함으로 설정하십시오. 『사용안함』이라는 함수 맵핑 옵션을 'y'(예, 이 함수 맵핑을 사용안합니다)로 설정하여 CREATE FUNCTION MAPPING문에서 그렇게 할 수 있습니다. 기본값은 'n'입니다.

Pass-Through를 사용하여 데이터 소스 직접 조회

*pass-through*라는 기능을 사용하여 데이터 소스에 고유한 SQL에서 데이터 소스를 조회할 수 있습니다. 이 절에서는

- 연합 서버와 연관 데이터 소스가 pass-through 세션에서 처리하는 SQL문의 종류를 진술합니다.
- pass-through를 사용할 때 인식해야 하는 고려사항과 제한사항을 나열합니다.

Pass-Through 세션으로 SQL 처리

다음 규칙에서는 SQL문을 DB2로 처리할지, 아니면 데이터 소스로 처리할지 여부를 지정합니다.

- 정적 명령문이 pass-through 세션에 제출되면 이는 연합 서버로 보내져서 처리됩니다.
- Pass-through 세션에서 SQL문을 데이터 소스로 제출하여 처리하려면, 이를 세션에서 동적으로 준비하고 세션이 열려 있을 때 실행해야 합니다.
  - SELECT문을 제출할 때 PREPARE문을 사용하여 준비한 다음 OPEN, FETCH 및 CLOSE문을 사용하여 조회의 결과를 액세스하십시오.
  - SELECT 외에 지원되는 명령문에 대해 다음과 같이 옵션이 두개 있습니다.
    - PREPARE문을 사용하여 지원되는 명령문을 준비하고 EXECUTE문을 사용하여 이를 실행하십시오.
    - EXECUTE IMMEDIATE문을 사용하여 준비하고 명령문을 실행하십시오.
- pass-through 세션 중 COMMIT 또는 ROLLBACK문을 사용할 경우 이 명령은 현재 작업 단위(UOW)를 완료합니다.

고려사항 및 제한사항

pass-through를 사용할 때 염두에 두어야 할 고려사항과 제한사항이 몇 가지 있습니다. 그 중 몇 가지는 일반 속성을 가지며 다른 사항은 Oracle 데이터 소스에만 적용됩니다.

모든 데이터 소스에 Pass-Through 사용

다음 정보는 모든 데이터 소스에 적용됩니다.

- Pass-through 세션에서 준비된 명령문은 동일한 pass-through 세션에서 실행되어야 합니다. Pass-through 세션에서 준비되지만, 동일한 pass-through 세션 밖에서 실행되는 명령문은 실패합니다(SQLSTATE 56098).
- Pass-through를 사용하여 데이터 소스를 기록할 수 있습니다. 예를 들어, 테이블 행을 삽입하고, 갱신하고, 삭제할 수 있습니다. 그러나, pass-through 세션의 UPDATE 및 DELETE문에서는 WHERE CURRENT OF 조건을 사용할 수 없다는 점에 주의하십시오.
- 응용프로그램은 다른 데이터 소스에 동일하게 영향을 주는 SET PASSTHRU 문을 몇 가지 포함할 수 있습니다. 응용프로그램에서 다중 SET PASSTHRU 문을 발행하더라도 pass-through 세션은 실제로 중첩되지 않습니다. 연합 서버

에서는 한 데이터 소스를 통과하여 다른 데이터 소스에 액세스하지 못합니다. 대신 서버에서 각 데이터 소스에 직접 액세스합니다.

- 여러 pass-through 세션이 동시에 열리면 각 세션에서 작업 단위(UOW)를 종료하려 할 때마다 COMMIT를 발행해야 합니다. 그런 다음 세션을 종료해야 할 경우 단일 SET PASSTHRU RESET문으로 그렇게 실행할 수 있습니다.
- pass-through 세션에서 SQL문에 정의된 호스트 변수는 :H*n*의 양식을 취해야 합니다. 여기서 H와 *n*은 고유한 정수입니다. *n*의 값은 0부터 연속으로 지정되어야 합니다.
- 한번에 하나 이상의 데이터 소스를 통과할 수 없습니다.
- Pass-through는 저장 프로시듀어 호출을 지원하지 않습니다.
- 통과는 SELECT INTO문을 지원하지 않습니다.

Oracle 데이터 소스에 Pass-Through 사용

다음 정보는 Oracle 데이터 소스에 적용됩니다.

- 원격 클라이언트에서 명령행 프로세서(CLIP)에서 pass-through 모드로 SELECT 문을 발행할 경우 다음 제한사항이 적용됩니다. 클라이언트 코드가 DB2 Universal Database 버전 5 이전의 DB2 응용프로그램 개발 클라이언트일 경우 SELECT에서 이유 코드 11로 SQLCODE -30090을 유도합니다. 이런 오류를 방지하려면 원격 클라이언트에서 버전 5 이상의 DB2 응용프로그램 개발 클라이언트를 사용해야 합니다.
- Oracle 서버에 대해 발행된 DDL문은 분석시 수행되므로 트랜잭션 구문론을 따르지 않습니다. 조작 완료시 Oracle에서 자동으로 조작을 확약합니다. 구간 복원이 발생할 경우 DDL은 구간 복원되지 않습니다.
- 원시 데이터 유형에서 SELECT문을 발행할 경우 RAWTOHEX 함수를 사용하여 16진 값을 수신하십시오. 원시 데이터 유형으로 INSERT를 수행할 때 16진 표시를 제공하십시오.

제6부 언어 고려사항

© Copyright IBM Corp. 1993, 2000

제20장 C 및 C++ 프로그래밍

C 및 C++에 대한 프로그래밍 고려사항 . . 661
C 및 C++의 언어 제한사항 661
C 및 C++의 Trigraph 절차 662
C++ 유형 장식물 고려사항 662
C 및 C++에 대한 입력과 출력 파일 . . . 663
C 및 C++에 대한 파일. 663
C 및 C++에 파일 포함. 667
C 및 C++에 SQL문 포함. 668
C 및 C++의 호스트 변수 670
C 및 C++의 호스트 변수 명명 670
C 및 C++의 호스트 변수 선언 671
C 및 C++의 표시기 변수 676
C 또는 C++의 그래픽 호스트 변수 선언 676
C 또는 C++의 LOB 데이터 선언 . . . 679
C 또는 C++의 LOB LOB 위치 지정자 선언 682
C 또는 C++의 파일 참조 선언 683
C 및 C++의 호스트 변수 초기화 . . . 684
C 매크로 예 684
C 및 C++의 호스트 구조 지원 685
C 및 C++의 표시기 테이블 687
C 및 C++의 널로 종료된 문자열 . . . 689
C 및 C++의 포인터 데이터 유형 . . . 690
C 및 C++에서 클래스 데이터 구성원을 호스트 변수로 사용 692
C 및 C++에서 구성 및 구성원 연산자 사용 693
C 및 C++에서의 그래픽 호스트 변수 처리 694
C 및 C++의 다중 바이트 문자 인코딩 694
C 및 C++에서 wchar_t 또는 sqldbchar 데이터 유형 선택 695
C 및 C++의 WCHARTYPE 사전 처리 컴파일러 옵션 696
C 및 C++의 일본어 또는 대만어 EUC 및 UCS-2 고려사항 700
C 및 C++의 지원되는 SQL 데이터 유형 701
C 및 C++의 2진 데이터용 708
C 및 C++의 SQLSTATE 및 SQLCODE 변수 708

C 및 C++에 대한 프로그래밍 고려사항

특수 호스트 언어 프로그래밍 고려사항은 다음 절에서 다룹니다. 언어 제한사항, 호스트 언어 고유의 include 파일, Embedded SQL문, 호스트 변수 및 호스트 변수에 대해 지원되는 데이터 유형에 대한 정보가 포함됩니다.

C 및 C++의 언어 제한사항

다음 절에서는 C/C++ 언어 제한사항에 대해 설명합니다.

© Copyright IBM Corp. 1993, 2000

C 및 C++의 Trigraph 절차

C 또는 C++ 문자 세트의 일부 문자가 모든 키보드에서 사용 가능한 것은 아닙니다. 이런 문자는 *trigraph*라 하는 세 개 문자의 절차를 사용하여 C 또는 C++ 소스 프로그램에 입력할 수 있습니다. SQL문으로는 trigraph를 인식하지 못합니다. 사전 처리 컴파일러는 호스트 변수 선언에서 다음 trigraph를 인식합니다.

Trigraph	정의
??(	왼쪽 각괄호 '['
??)	오른쪽 각괄호 ']'
??<	왼쪽 대괄호 '{'
??>	오른쪽 대괄호 '}'

아래 나열되는 나머지 trigraph는 C 또는 C++ 소스 프로그램의 다른 곳에서 나타날 수도 있습니다.

Trigraph	정의
??=	해쉬 기호 '#'
??/	역슬래시 '\'
??'	탈자 기호 '^'
??!	수직선 '|'
??–	틸데 '~'

C++ 유형 장식물 고려사항

C++를 사용하여 저장 프로시듀어나 UDF를 작성할 때 프로시듀어나 UDF 선언을 고려할 수 있습니다.

```
extern "C" ...procedure or function declaration...
```

extern "C"는 C++ 컴파일러의 함수 이름 유형 장식물을 방지합니다. 이런 선언이 없으면 저장 프로시듀어를 호출하거나 CREATE FUNCTION문을 발행할 때 함수 이름에 대한 모든 유형 장식물을 포함해야 합니다.

C 및 C++에 대한 입력과 출력 파일

기본적으로 입력 파일의 확장자는 다음과 같습니다.

.sqc 지원되는 모든 플랫폼에서의 C 파일

.sqC UNIX 플랫폼에서의 C++ 파일

.sqx OS/2 및 Windows 32 운영 체제에서의 C++ 파일

기본적으로, 해당 사전 처리 컴파일러 출력 파일의 확장자는 다음과 같습니다.

.c 지원되는 모든 플랫폼에서의 C 파일

.C UNIX 플랫폼에서의 C++ 파일

.cxx OS/2 및 Windows 32 운영 체제에서의 C++ 파일

OUTPUT 사전 처리 컴파일을 사용하여 출력 수정 소스 파일의 이름과 경로를 겹쳐쓸 수 있습니다. TARGET C나 TARGET CPLUSPLUS 사전 처리 컴파일 옵션을 사용할 경우 입력 파일에 특정 확장자는 필요하지 않습니다.

C 및 C++에 대한 파일

C 및 C++용 호스트 언어 고유의 include 파일(헤더 파일)의 파일 확장자는 `.h`입니다. 사용자의 응용프로그램에 사용할 include 파일은 아래에서 설명합니다.

SQL (sql.h)
이 파일에는 바인더, 사전 처리 컴파일러에 대한 언어 특정 프로토타입과 오류 메시지 API가 있습니다. 또한 시스템 상수를 정의합니다.

SQLADEF (sqladef.h)
이 파일에는 사전 처리 컴파일된 C 및 C++ 응용프로그램에서 사용하는 함수 프로토타입이 있습니다.

SQLAPREP (sqlaprep.h)
이 파일에는 사용자의 사전 처리 컴파일러를 작성하기 위해 요구되는 정의가 있습니다.

SQLCA (sqlca.h)
이 파일은 SQLCA 구조를 정의합니다. SQLCA에는 SQL문과 API 호출

의 실행에 관한 오류 정보와 함께 응용프로그램을 제공할 데이터베이스 관리 프로그램에 의해 사용되는 변수가 있습니다.

SQLCLI (sqlcli.h)

이 파일에는 DB2 CLI(Call Level Interface) 응용프로그램을 작성하는 데 필요한 함수 프로토타입과 상수가 있습니다. 이 파일의 함수는 X/Open 콜 레벨 인터페이스와 ODBC 코어 레벨에 공통됩니다.

SQLCLI1 (sqlcli1.h)

이 파일에는 DB2 CLI에서 고급 기능을 활용하는 DB2 CLI를 작성하는 데 필요한 함수 프로토타입과 상수가 있습니다. 이 파일의 대부분 함수는 X/Open 콜 레벨 인터페이스와 ODBC 코어 레벨에 공통됩니다. 또한 이 파일에는 X/Open 전용 함수와 DB2 고유 함수도 있습니다.

이 파일에는 `sqlcli.h`와 `sqlext.h`(ODBC 레벨 2 API 정의를 포함한)가 있습니다.

SQLCODES (sqlcodes.h)

이 파일은 SQLCA 구조의 SQLCODE에 대한 상수를 정의합니다.

SQLDA (sqlda.h)

이 파일은 SQLDA 구조를 정의합니다. SQLDA는 응용프로그램과 데이터베이스 관리 프로그램 사이에서 데이터를 전달하기 위해 사용됩니다.

SQLEAU (sqleau.h)

이 파일에는 DB2 보안 감사 API에 요구되는 상수 및 구조 정의가 있습니다. 이러한 API를 사용하면 프로그램에서 이 파일을 포함시킬 필요가 있습니다. 또한 이 파일에는 감사 추적 레코드의 필드에 대한 상수 및 키워드 값 정의가 포함됩니다. 외부 또는 벤더 감사 추적 추출 프로그램에서 이런 정의를 사용할 수 있습니다.

SQLENV (sqlenv.h)

이 파일은 데이터베이스 환경 API용 언어 특정 호출과 그러한 인터페이스에 대한 구조, 상수 및 리턴 코드를 정의합니다.

SQLEXT (sqlext.h)

이 파일에는 X/Open 콜 레벨 인터페이스 스펙의 일부가 아니므로

Microsoft Corporation의 허가 아래 사용하는 ODBC 레벨 1과 레벨 2 API의 함수 프로토타입과 상수가 있습니다.

SQLE819A (sqle819a.h)

데이터베이스의 코드 페이지가 819(ISO Latin-1)일 경우, 이 순서는 호스트 CCSID 500(EBCDIC 국제어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLE819B (sqle819b.h)

데이터베이스의 코드 페이지가 819(ISO Latin-1)일 경우, 이 순서는 호스트 CCSID 037(EBCDIC 미국 영어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLE850A (sqle850a.h)

데이터베이스의 코드 페이지가 850(ASCII Latin-1)일 경우, 이 순서는 호스트 CCSID 500(EBCDIC 국제어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLE850B (sqle850b.h)

데이터베이스의 코드 페이지가 850(ASCII Latin-1)일 경우, 이 순서는 호스트 CCSID 037(EBCDIC 미국 영어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLE932A (sqle932a.h)

데이터베이스의 코드 페이지가 932(ASCII Japanese)일 경우, 이 순서는 호스트 CCSID 5035(EBCDIC 일본어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLE932B (sqle932b.h)

데이터베이스의 코드 페이지가 932(ASCII Japanese)일 경우, 이 순서는

호스트 CCSID 5026(EBCDIC 일본어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLJACB (sqljacb.h)

이 파일에서는 DB2 Connect 인터페이스용 상수, 구조체 및 제어 블록을 정의합니다.

SQLMON (sqlmon.h)

이 파일은 데이터베이스 시스템 모니터 API용 언어 특정 호출과 그러한 인터페이스에 대한 구조, 상수 및 리턴 코드를 정의합니다.

SQLSTATE (sqlstate.h)

이 파일은 SQLCA 구조의 SQLSTATE 필드에 대한 상수를 정의합니다.

SQLSYSTM (sqlsystm.h)

이 파일에는 데이터베이스 관리 프로그램 API 및 데이터 구조에 의해 사용된 플랫폼 특정 정의가 있습니다.

SQLUDF (sqludf.h)

이 파일에서는 사용자 정의 함수(UDF) 작성시 사용되는 상수와 인터페이스 구조체를 정의합니다. 이 파일에 대해서는 465 페이지의『UDF include 파일: sqludf.h』에서 자세한 내용을 참조하십시오.

SQLUTIL (sqlutil.h)

이 파일은 유틸리티 API용 언어 특정 호출과 그러한 인터페이스에 요구되는 구조, 상수 및 코드를 정의합니다.

SQLUV (sqluv.h)

이 파일에서는 비동기 Read Log API와, 테이블 로드 및 로드 해제 벤더에서 사용하는 API용 구조체, 상수 및 프로토타입을 정의합니다.

SQLUVEND (sqluvend.h)

이 파일에서는 저장영역 관리 벤더에서 사용할 API용 구조체, 상수 및 프로토타입을 정의합니다.

SQLXA (sqlxa.h)

이 파일에는 X/Open XA 인터페이스를 사용하는 응용프로그램용 함수 프로토타입과 상수가 있습니다.

C 및 C++에 파일 포함

파일을 포함하는 방법은 EXEC SQL INCLUDE문 사용과 #include 매크로 사용의 두 가지 방법이 있습니다. 사전 처리 컴파일러에서는 #include를 무시하고, EXEC SQL INCLUDE문으로 포함한 파일만을 처리합니다.

EXEC SQL INCLUDE를 사용하여 포함한 파일을 찾기 위해 DB2 C 사전 처리 컴파일러에서는 먼저 현재 디렉토리를 검색한 다음 DB2INCLUDE 환경 변수로 지정한 디렉토리를 검색합니다. 다음 예를 고려하십시오.

- `EXEC SQL INCLUDE payroll;`

  위와 같이 INCLUDE문에 지정된 파일을 인용 부호로 닫지 않은 경우 C 사전 처리 컴파일러는 자신이 찾은 각 디렉토리에서 `payroll.sqc`를 탐색한 다음 `payroll.h`를 검색합니다. UNIX 운영 체제의 경우 C++ 사전 처리 컴파일러는 자신이 찾은 각 디렉토리에서 `payroll.sqC`, `payroll.sqx`, `payroll.hpp`, `payroll.h` 순서로 검색합니다. OS/2 또는 Windows-32 비트 운영 체제의 경우 C++ 사전 처리 컴파일러는 자신이 찾은 각 디렉토리에서 `payroll.sqx`, `payroll.hpp`, `payroll.h` 순서로 검색합니다.

- `EXEC SQL INCLUDE 'pay/payroll.h';`

  위와 같이 파일 이름을 인용 부호로 닫을 경우 확장자는 이름에 추가되지 않습니다.

  인용 부호 안의 파일 이름에 절대 경로가 없을 경우 INCLUDE 파일 이름에 지정된 경로에 따라 DB2INCLUDE의 내용을 사용하여 파일을 검색합니다. 예를 들어, UNIX 기반 시스템의 경우 DB2INCLUDE가 '`/disk2:myfiles/c`'에 설정된 경우 /C++ 사전 처리 컴파일러에서 '`./pay/payroll.h`', '`/disk2/pay/payroll.h`', 마지막으로 '`./myfiles/c/pay/payroll.h`'를 검색합니다. 실제로 파일이 있는 경로가 사전 처리 컴파일러 메시지에 표시됩니다. OS/2 및 Windows 기반 운영 체제의 경우 위 예에서 정방향 슬래시를 역슬래시(\)로 대체하십시오.

주: DB2 명령행 처리기에서 DB2INCLUDE의 설정을 캐쉬화합니다. CLP 명령을 발행한 후 DB2INCLUDE의 설정을 변경하려면 TERMINATE 명령을 입력한 다음 데이터베이스에 다시 연결하고 평소와 마찬가지로 사전 처리 컴파일하십시오.

컴파일러 오류를 다시 원래 소스와 관련시키기 위해 사전 처리 컴파일러에서 출력 파일에 ANSI #line 매크로를 생성합니다. 이를 통해 컴파일러에서 사전 처리 컴파일러 출력 대신에 파일 이름과 소스 또는 포함된 소스 파일의 행 수를 사용하여 오류를 보고할 수 있습니다.

그러나 PREPROCESSOR option 옵션을 지정할 경우 사전 처리 컴파일러에서 생성한 모든 #line 매크로는 외부 C 프로세서의 선행 처리 파일을 참조합니다. PREPROCESSOR 옵션에 대해서는 684 페이지의 『C 매크로 예』의 내용을 참조하십시오.

소스 코드와 목표 코드를 관련시키는 일부 디버거와 기타 도구에서 #line 매크로에 대한 작업이 항상 원활하지는 않습니다. 사용하려는 도구가 예상과 달리 수행될 경우 사전 컴파일시 NOLINEMACRO 옵션(DB2 PREP와 함께 사용)을 사용하십시오. 그렇게 하면 #line 매크로 생성을 방지할 수 있습니다.

C 및 C++에 SQL문 포함

Embedded SQL문은 다음 세 구성요소로 구성됩니다.

구성요소	올바른 구문
명령문 초기화 프로그램	EXEC SQL
명령문 문자열	유효한 SQL문
명령문 종료 표시자	세미콜론(;)

예를 들면 다음과 같습니다.

```
EXEC SQL SELECT col INTO :hostvar FROM table;
```

다음 규칙이 Embedded SQL문에 적용됩니다.

- 키워드 쌍이나 분리 행과 같은 행에서 SQL문 문자열을 시작할 수 있습니다. 명령문 문자열의 길이는 몇 행이 될 수 있습니다. 행 사이의 EXEC SQL 키워드 쌍을 분리하지 마십시오.
- SQL문 종료 표시자를 사용해야 합니다. 종료 표시자를 사용하지 않으면 사전 처리 컴파일러가 응용프로그램에서 다음 종료 표시자가 나타날 때까지 계속 실행됩니다. 따라서 종료 오류가 발생할 수 있습니다.

C/C++ 주석은 명령문 초기화 프로그램 앞이나 명령문 종료 표시자 뒤에 지정할 수 있습니다.

- 같은 행에 SQL문과 C/C++문을 여러 개 지정할 수 있습니다. 예를 들면 다음과 같습니다.

```
EXEC SQL OPEN c1; if (SQLCODE >= 0) EXEC SQL FETCH c1 INTO :hv;
```

- SQL 사전 처리 컴파일러는 인용 부호로 싸인 문자열의 CR/LF와 TAB를 그대로 둡니다.
- Embedded SQL문의 일부인 모든 행에서 SQL 주석이 허용됩니다. 동적으로 실행되는 명령문에서는 이런 주석이 허용되지 않습니다. SQL 주석의 형식은 이중 대시(--) 다음에 0개 이상의 문자로 구성되며 행의 끝으로 종료됩니다. C/C++ 언어의 일부인 것처럼 보여 컴파일 오류가 발생되므로 SQL문 종료 표시자 뒤에 SQL문을 지정하지 마십시오.

  공백이 허용되는 정적 문자열에서 주석을 사용할 수 있습니다. C/C++ 주석 분리문자 /* */나 SQL 주석 기호(--)를 사용하십시오. 정적 SQL문에 // 스타일 C++ 주석은 허용되지 않지만 프로그램이 다른 곳에서는 사용할 수 있습니다. 사전 처리 컴파일러에서 SQL문을 처리하기 전에 주석을 제거합니다. 동적 SQL문에서 C 및 C++ 주석 분리문자 /* */나 //를 사용할 수 **없습니다**. 그러나 프로그램의 다른 곳에서는 사용할 수 있습니다.

- C 및 C++ 응용프로그램의 행 분리 위에 SQL 문자열 리터럴과 분리 식별자를 계속 사용할 수 있습니다. 그렇게 하려면 분리하려는 행의 끝에 역슬래시(\)를 사용하십시오. 예를 들면 다음과 같습니다.

```
EXEC SQL SELECT "NA\
ME" INTO :n FROM staff WHERE name='Sa\
nders';
```

  캐리지 리턴이나 라인 피드와 같은 새로운 행 문자는 문자열이나 분리 식별자에 포함되지 않습니다.

- 행의 끝(eol)과 TAB 문자와 같은 공백 문자 대체는 다음과 같이 발생합니다.
  - SQL문 안이지만 외부 인용 부호 외부에 나타날 경우 행의 끝과 TAB은 단일 공백으로 대체됩니다.
  - 인용 부호 안에서 나타날 때 C 프로그램에서 문자열이 제대로 연속되지 않을 경우 문자열이 행의 끝 문자가 사라집니다. TAB은 수정되지 않습니다.

행의 끝과 TAB에 사용되는 실제 문자는 플랫폼에 따라 달라진다는 점에 주의하십시오. 예를 들어, OS/2에서는 행의 끝에 대해 캐리지 리턴이나 라인 피드를 사용하지만 UNIX 기반 시스템에서는 라인 피드만 사용합니다.

C 및 C++의 호스트 변수

호스트 변수는 SQL문에서 참조되는 C 또는 C++ 언어 변수입니다. 이런 변수를 사용하여 응용프로그램에서 입력 데이터를 전달하고 데이터베이스 관리 프로그램으로부터 출력 데이터를 수신할 수 있습니다. 응용프로그램을 사전 처리 컴파일하면 컴파일러에서 다른 C/C++ 변수와 마찬가지로 호스트 변수를 사용합니다. 다음 규칙은 호스트 변수를 명명, 선언 그리고 사용할 때 다음 절에서 설명됩니다.

C 및 C++의 호스트 변수 명명

SQL 사전 처리 컴파일러에서 선언된 이름으로 호스트 변수를 식별합니다. 다음 규칙이 적용됩니다.

- 최대 255자까지 변수 이름을 지정하십시오.
- 시스템용으로 예약된 SQL, sql, DB2 및 db2, 이외의 접두부로 호스트 변수를 시작하십시오. 예를 들면 다음과 같습니다.

```
EXEC SQL BEGIN DECLARE SECTION;
     char  varsql;     /* allowed */
     char  sqlvar;     /* not allowed */
     char  SQL_VAR;    /* not allowed */
EXEC SQL END DECLARE SECTION;
```

- 사전 처리 컴파일러는 호스트 변수 이름을 모듈의 전역으로 간주합니다. 그러나 호스트 변수를 전역 변수로 선언해야 한다는 의미는 아닙니다. 함수 내에서 호스트 변수를 지역 변수로 선언하는 것은 좋습니다. 예를 들어, 다음 코드가 제대로 작동합니다.

```
   void f1(int i)
   {
EXEC SQL BEGIN DECLARE SECTION;
     short host_var_1;
EXEC SQL END DECLARE SECTION;
   EXEC SQL SELECT COL1 INTO :host_var_1 from TBL1;
   }
   void f2(int i)
   {
```

```
EXEC SQL BEGIN DECLARE SECTION;
     short host_var_2;
EXEC SQL END DECLARE SECTION;
   EXEC SQL INSERT INTO TBL1 VALUES (:host_var_2);
   }
```

또한 유형과 크기가 같은 한 몇 개의 지역 호스트 변수는 같은 이름을 가질 수 있습니다. 그렇게 하려면 호스트 변수의 첫번째 발생을 BEGIN DECLARE SECTION과 END DECLARE SECTION 문 사이에서 사전 처리 컴파일러에 선언하고, 선언 절에서 변수에 대한 후속 선언을 선언 절에서 제거하십시오. 다음 코드가 그 예를 나타냅니다.

```
   void f3(int i)
   {
EXEC SQL BEGIN DECLARE SECTION;
   char host_var_3[25];
EXEC SQL END DECLARE SECTION;
   EXEC SQL SELECT COL2 INTO :host_var_3 FROM TBL2;
   }
   void f4(int i)
   {
   char host_var_3[25];
   EXEC SQL INSERT INTO TBL2 VALUES (:host_var_3);
   }
```

f3과 f4가 같은 모듈에 있고 두 함수에서 host_var_3의 유형과 길이가 같을 경우 사전 처리 컴파일러에 대한 단일 선언은 두 위치에서 사용하기에 충분합니다.

C 및 C++의 호스트 변수 선언

SQL 선언 절은 호스트 변수 선언을 식별하는 데 사용해야 합니다. 이 절은 후속 SQL문에서 참조할 수 있는 호스트 변수를 후속 사전 처리 컴파일러에 알려 줍니다.

C/C++ 사전 처리 컴파일러에서는 유효한 C 또는 C++ 선언의 부속 집합을 유효한 호스트 변수 선언으로만 인식합니다. 이런 선언에서 숫자나 문자 변수를 정의합니다. 호스트 변수의 Typedefs는 허용되지 않습니다. 호스트 변수는 단일 호스트 구조로 그룹화할 수 있습니다. 호스트 구조에 대해서는 685 페이지의 『C 및 C++의 호스트 구조 지원』의 내용을 참조하십시오. C++ 클래스 데이터 구성원을

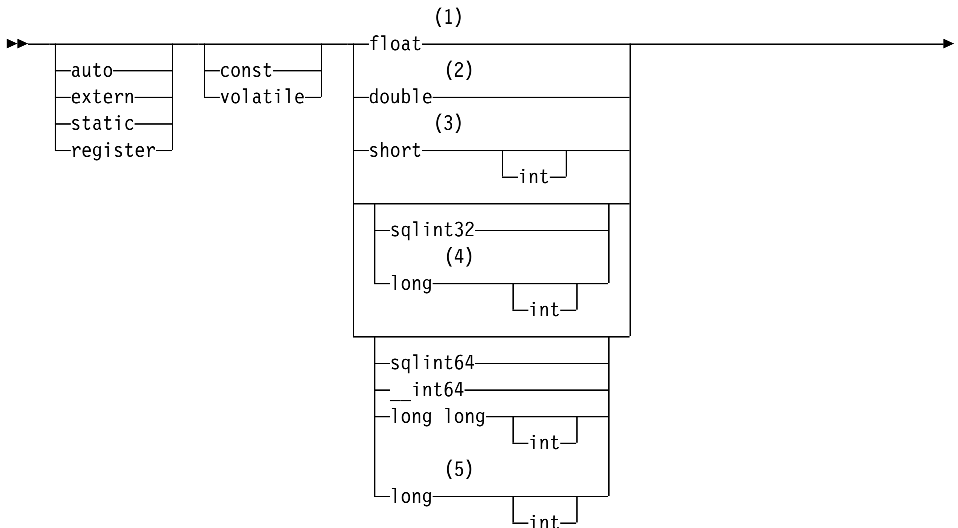
호스트 변수로 정의할 수 있습니다. 클래스에 대해서는 692 페이지의 『C 및 C++에서 클래스 데이터 구성원을 호스트 변수로 사용』의 내용을 참조하십시오.

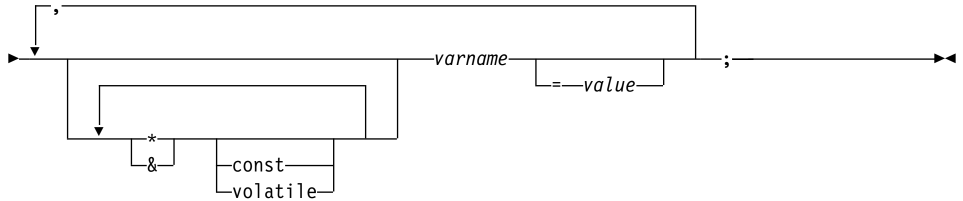
숫자 호스트 변수는 숫자 SQL 입력 또는 출력 값의 입력 또는 출력 변수로 사용할 수 있습니다. 문자 호스트 변수는 문자, 날짜, 시간 또는 시간소인 SQL 입력 또는 출력 값의 입력 또는 출력 변수로 사용할 수 있습니다. 응용프로그램에서 출력 변수가 수신한 값을 포함할만큼 충분히 긴 지 확인해야 합니다.

구조화 유형에 대해 호스트 변수를 선언하는 것에 대한 정보는 381 페이지의 『구조화 유형 호스트 변수 선언』을 참조하십시오.

C 또는 C++에서 숫자 호스트 변수의 구문에서 C 또는 C++로 숫자 호스트 변수를 선언하는 구문을 나타냅니다.

C 또는 C++에서 숫자 호스트 변수의 구문





주:

1 REAL (SQLTYPE 480), 길이 4

2 DOUBLE (SQLTYPE 480), 길이 8

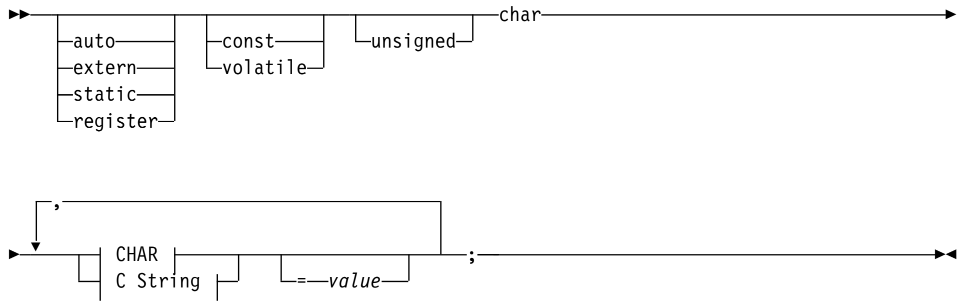
3 SMALLINT (SQLTYPE 500)

4 최대 응용프로그램 이식성을 위해 INTEGER 및 BIGINT 호스트 변수인 경우에는 sqlint32와 sqlint64를 사용하십시오. 기본적으로, 긴 호스트 변수를 사용하면, 64 BIT UNIX와 같은 64비트 수량 플랫폼에서 사전 처리 컴파일 오류 SQL0402가 발생할 수 있습니다. PREP 옵션 LONGERROR NO를 사용하여 DB2에서 긴 변수를 승인 가능한 호스트 변수 유형으로 승인하게 만들고 이를 BIGINT 변수로 처리하십시오.

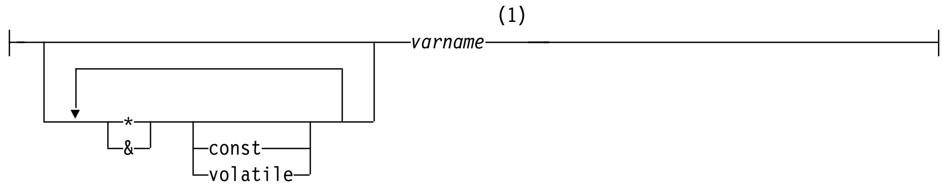
5 최대 응용프로그램 이식성을 위해 INTEGER 및 BIGINT 호스트 변수인 경우에는 sqlint32와 sqlint64를 사용하십시오. BIGINT 데이터 유형을 사용하려면 플랫폼에서 64비트 정수 값을 지원해야 합니다. 기본적으로, 긴 호스트 변수를 사용하면, 64 BIT UNIX와 같은 64비트 수량 플랫폼에서 사전 처리 컴파일 오류 SQL0402가 발생할 수 있습니다. PREP 옵션 LONGERROR NO를 사용하여 DB2에서 긴 변수를 승인 가능한 호스트 변수 유형으로 승인하게 만들고 이를 BIGINT 변수로 처리하십시오.

양식 1: C/C++에서 고정 길이의 널로 종료되는 문자 호스트 변수의 구문은 C 또는 C++에서 고정 길이의 널로 종료되는(null-terminated) 문자 호스트 변수에 대한 구문을 나타냅니다.

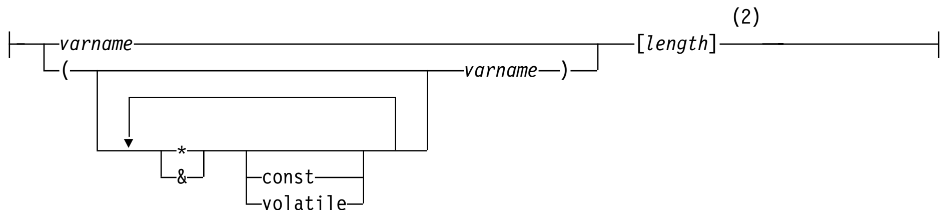
양식 1: C/C++에서 고정 길이의 널로 종료되는 문자 호스트 변수의 구문



CHAR



C String



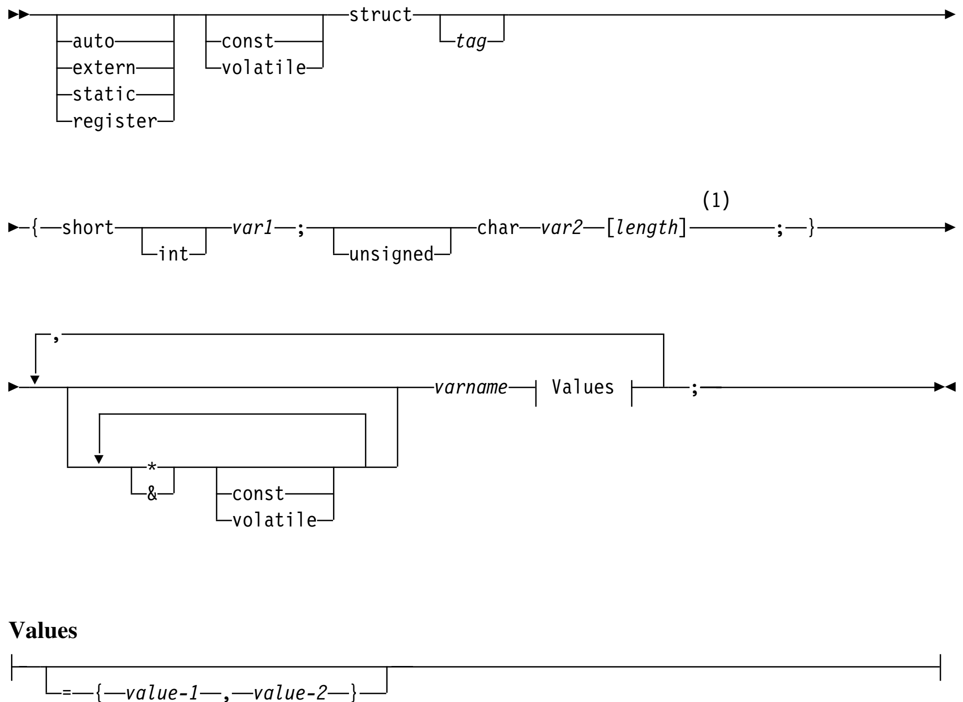
주:

1 CHAR (SQLTYPE 452), length 1

2 널로 종료되는 C 문자열 (SQLTYPE 460); 길이는 유효한 상수 표현식이 될 수 있습니다.

양식 2: C/C++에서 변수 길이 문자 호스트 변수의 구문은 C 또는 C++에서 변수 길이 문자 호스트 변수를 선언하는 구문을 나타냅니다.

양식 2: C/C++에서 변수 길이 문자 호스트 변수의 구문



주:

1 형식 2에서 길이는 유효한 상수 표현식이 될 수 있습니다. 평가 뒤의 값으로 호스트 변수가 VARCHAR(SQLTYPE 448) 또는 LONG VARCHAR (SQLTYPE 456)인지 판별합니다.

변수 길이 문자 호스트 변수 고려사항:

1. 데이터베이스 관리 프로그램에서 가능한 때 문자 데이터를 **양식 1**이나 **양식 2**로 변환한지만 **양식 1**은 CHAR 또는 VARCHAR 컬럼 유형에 대응하지만 **양식 2**는 VARCHAR 및 LONG VARCHAR 컬럼 유형에 대응합니다.
2. **양식 1**에 길이 지정자 *[n]*을 사용할 경우 평가 뒤 길이 지정자 값은 32672보다 작거나 같아야 하며 변수에 포함되는 문자열은 널로 종료되어야 합니다.
3. **양식 2**를 사용할 경우 평가 뒤 길이 지정자의 값은 32 700보다 작거나 같아야 합니다.
4. **양식 2**에서 *var1*과 *var2*은 연산자가 없는 단순 변수 참조여야 하며 호스트 변수(*varname*이 호스트 변수)로 사용할 수 없습니다.

5. *varname*은 단순 변수 이름이거나 **varname*과 같은 연산자를 포함할 수 있습니다. 690 페이지의 『C 및 C++의 포인터 데이터 유형』에서 자세한 내용을 참조하십시오.
6. 사전 처리 컴파일러에서 모든 호스트 변수의 SQLTYPE와 SQLLEN을 판별합니다. 호스트 변수가 표시기 변수가 포함된 SQL문에 나타나면 그 명령문 지속기간 중 기본 SQLTYPE+1이 되도록 할당됩니다.
7. 사전 처리 컴파일러는 C 또는 C++에서 구문상으로 유효하지 않은 선언을 몇 가지 허용합니다. 특정 선언 구문에 대해서 궁금한 점이 있으면 컴파일러 설명서를 참조하십시오.

C 및 C++의 표시기 변수

표시기 변수는 `short` 데이터 유형으로 선언해야 합니다.

C 또는 C++의 그래픽 호스트 변수 선언

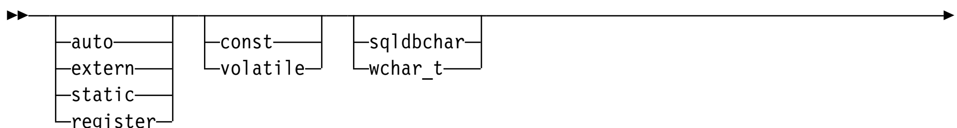
그래픽 호스트 변수 선언은 다음 세 양식 중 하나를 취할 수 있습니다.

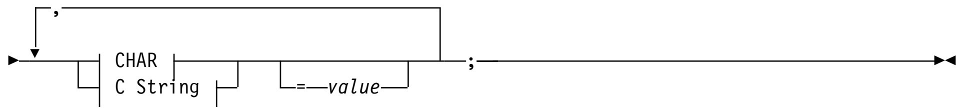
- 단일 그래픽 양식
- 널로 종료되는 그래픽 양식
- VARGRAPHIC 구조 양식

그래픽 호스트 변수 사용에 대해서는 694 페이지의 『C 및 C++에서의 그래픽 호스트 변수 처리』에서 자세한 내용을 참조하십시오.

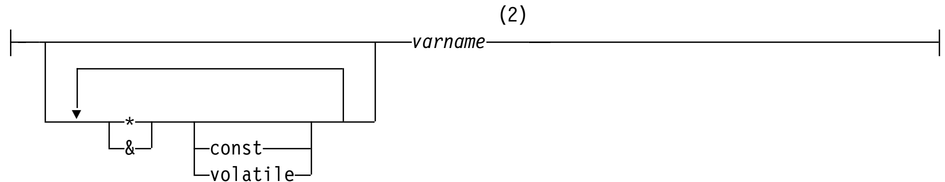
그래픽 선언 구문(단일 그래픽 양식과 널로 종료되는 그래픽 양식)은 단일 그래픽 양식과 널로 종료되는 그래픽 양식을 사용하여 그래픽 호스트 변수를 선언하는 구문을 나타냅니다.

그래픽 선언 구문(단일 그래픽 양식과 널로 종료되는 그래픽 양식)

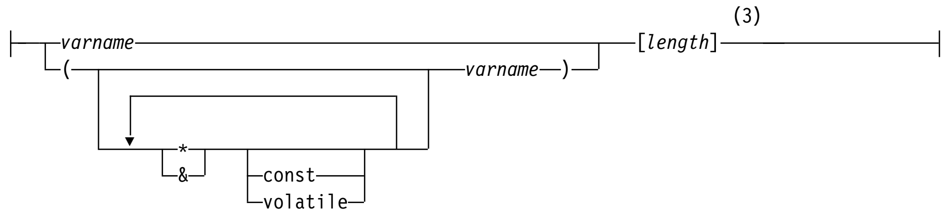




CHAR



C String



주:

1 두 그래픽 유형 중 사용할 유형을 판별하려면 695 페이지의 『C 및 C++에서 wchar_t 또는 sqldbchar 데이터 유형 선택』에서 자세한 내용을 참조하십시오.

2 GRAPHIC (SQLTYPE 468), length 1

3 널로 종료되는 그래픽 문자열(SQLTYPE 400)

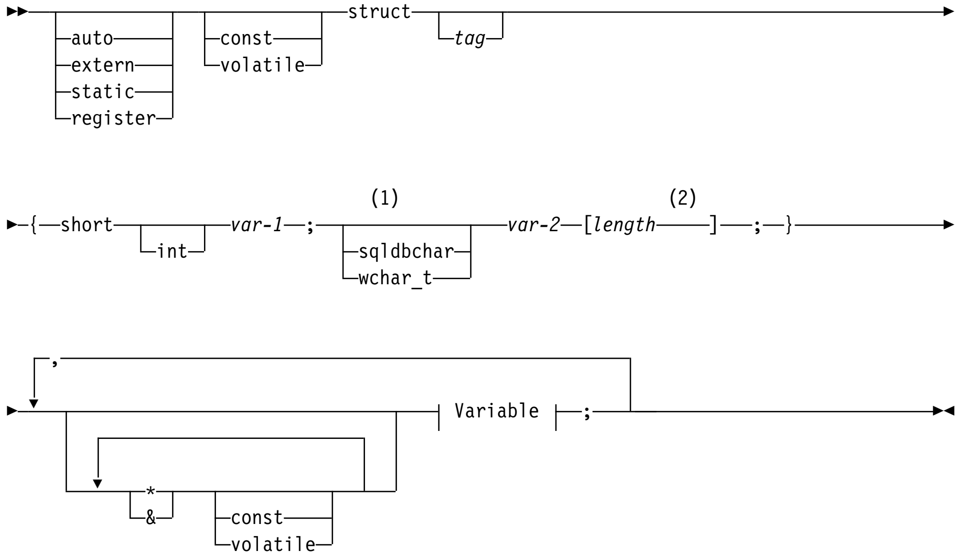
그래픽 호스트 변수 고려사항:

1. 단일 그래픽 양식에서 468 또는 469의 SQLTYPE로, 길이가 1인 고정 길이 그래픽 문자열을 선언합니다.
2. *value*는 초기 설정자입니다. WCHARTYPE CONVERT 옵션을 사용할 경우 와이드 문자 문자열 리터럴(L-리터럴)을 사용해야 합니다.
3. *length*는 유효한 상수 표현식이 될 수 있으며, 평가 뒤 그 값은 1보다 크거나 같아야 하며 VARGRAPHIC의 최대 길이 16 336보다 작아야 합니다.

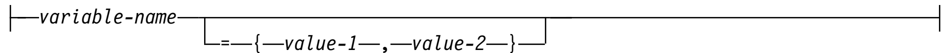
4. 널로 종료되는 그래픽 문자열은 표준 레벨 사전 처리 컴파일 옵션 설정의 값에 따라 다르게 조절됩니다. 689 페이지의 『C 및 C++의 널로 종료된 문자열』에서 자세한 내용을 참조하십시오.

그래픽 선언의 구문(VARGRAPHIC 구조 양식)은 VARGRAPHIC 구조 양식을 사용하여 그래픽 호스트 변수를 선언하는 구문을 나타냅니다.

그래픽 선언의 구문(VARGRAPHIC 구조 양식)



Variable:



주:

1 두 그래픽 유형 중 사용할 유형을 판별하려면 695 페이지의 『C 및 C++에서 wchar_t 또는 sqldbchar 데이터 유형 선택』에서 자세한 내용을 참조하십시오.

2 길이는 유효한 상수 표현식이 될 수 있습니다. 평가 뒤의 값으로 호스트 변수가 VARGRAPHIC(SQLTYPE 464) 또는 LONG VARGRAPHIC (SQLTYPE 472)인지 판별합니다. 길이 값은 1과 같거나 커야 하며 LONG VARGRAPHIC의 최대 길이(16350)보다 작아야 합니다.

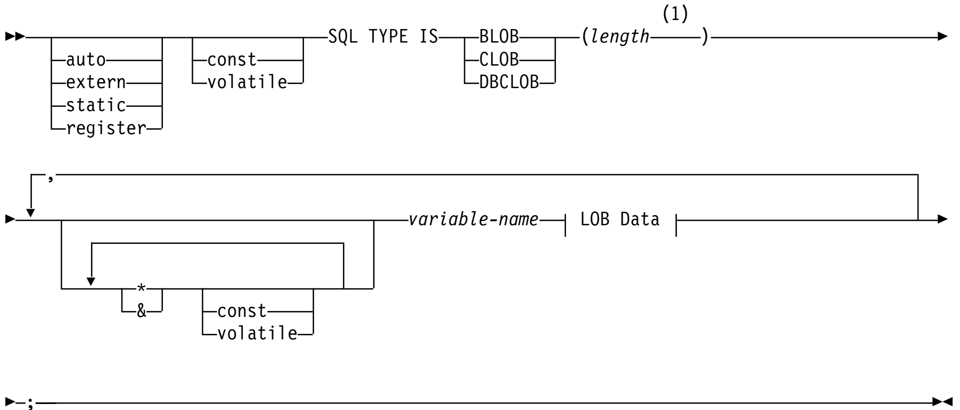
그래픽 선언(VARGRAPHIC 구조 양식) 고려사항:

1. *var-1* 및 *var-2*는 연산자가 없는 단순 변수 참조여야 하며 호스트 변수로 사용할 수 없습니다.
2. *value-1* 및 *value-2*는 *var-1* 및 *var-2*의 초기 설정자입니다. *value-1*은 정수여야 하고 WCHARTYPE CONVERT 사전 처리 컴파일러 옵션을 사용할 경우 *value-2*는 와이드 문자 문자열 리터럴(L-리터럴)이어야 합니다.
3. *tag* 구조는 다른 데이터 영역을 정의하는 데 사용할 수 있지만 그 자체는 호스트 변수로 사용할 수 없습니다.

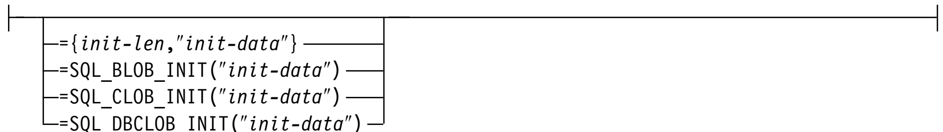
C 또는 C++의 LOB 데이터 선언

C/C++에서 대형 오브젝트(LOB) 호스트 변수의 구문은 C 또는 C++에서 대형 오브젝트(LOB) 호스트 변수를 선언하는 구문을 나타냅니다.

C/C++에서 대형 오브젝트(LOB) 호스트 변수의 구문



LOB Data



주:

1 길이는 유효한 상수 표현식이 될 수 있으며 여기서 K, M 또는 G 상수를 사

용할 수 있습니다. BLOB 및 CLOB를 평가한 후 길이 값은 1 <= *length* <= 2 147 483 647이어야 합니다. DBCLOB에 대해 평가한 후 길이 값은 1 <= length <= 1 073 741 823이어야 합니다.

LOB 호스트 변수 고려사항:

1. 함수로 전달되는 LOB 유형 호스트 변수에 대해 유형 점검과 함수 결정을 실행할 수 있도록 세 가지 LOB 유형을 서로 구별하려면 SQL TYPE IS절이 필요합니다.
2. SQL TYPE IS, BLOB, CLOB, DBCLOB, K, M, G는 대소문자 혼합으로 나타날 수 있습니다.
3. 초기 문자열 *"init-data"*에 허용된 최대 길이는 문자열 분리문자를 포함하는 32702 바이트(사전 처리 컴파일러에서 C/C++ 문자열의 기존 한계와 동일)입니다.
4. 초기 길이 *init-len*은 숫자 상수여야 합니다. 즉, K, M 또는 G를 포함할 수 없습니다.
5. LOB의 길이는 지정해야 합니다. 즉, 다음 선언은 허용되지 않습니다.

   ```
   SQL TYPE IS BLOB my_blob;
   ```

6. 선언에서 LOB를 초기화하지 않은 경우 사전 처리 컴파일러 생성 코드에서 초기화는 실행되지 않습니다.
7. DBCLOB를 초기화한 경우 사용자가 문자열의 접두부로 'L'(와이드 문자 문자열 지시)을 지정해야 합니다.

   주: WCHARTYPE CONVERT 사전 처리 컴파일 옵션을 선택한 경우 사전 처리 컴파일 프로그램에서 `L"Hello"`와 같은 와이드 문자 리터럴만 사용해야 합니다.

8. 사전 처리 컴파일러에서 호스트 변수의 유형으로 변환하는 데 사용할 수 있는 구조 태그를 생성합니다.

BLOB 예:

선언:

```
static Sql Type is Blob(2M) my_blob=SQL_BLOB_INIT("mydata");
```

그 결과 다음 구조가 생성됩니다.

```
static struct my_blob_t {
     sqluint32       length;
     char            data[2097152];
 } my_blob=SQL_BLOB_INIT("mydata");
```

CLOB 예:

선언:

```
volatile sql type is clob(125m) *var1, var2 = {10, "data5data5"};
```

그 결과 다음 구조가 생성됩니다.

```
volatile struct var1_t {
     sqluint32       length;
     char            data[131072000];
} * var1, var2 = {10, "data5data5"};
```

DBCLOB 예:

선언:

```
SQL TYPE IS DBCLOB(30000) my_dbclob1;
```

WCHARTYPE NOCONVERT 옵션으로 사전 처리 컴파일하면 다음 구조가 생성됩니다.

```
struct my_dbclob1_t {
     sqluint32       length;
     sqldbchar       data[30000];
} my_dbclob1;
```

선언:

```
SQL TYPE IS DBCLOB(30000) my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

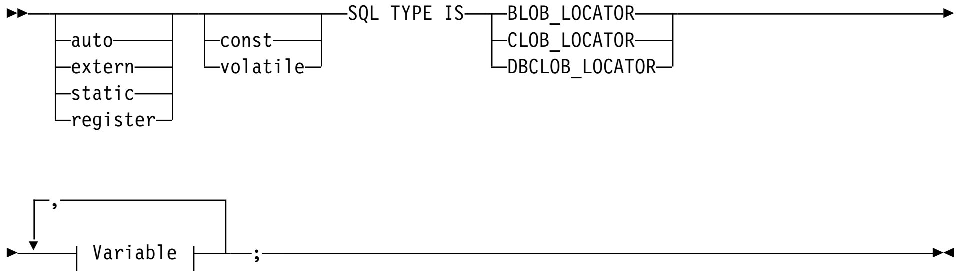
WCHARTYPE CONVERT 옵션으로 사전 처리 컴파일하면 다음 구조가 생성됩니다.

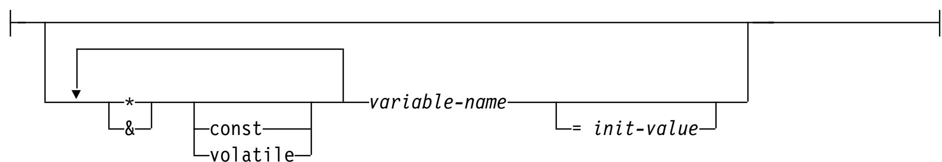
```
struct my_dbclob2_t {
     sqluint32       length;
     wchar_t         data[30000];
} my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

C 또는 C++의 LOB LOB 위치 지정자 선언

C/C++에서 대형 오브젝트(LOB) 위치 지정자 호스트 변수의 구문은 C 또는 C++에서 대형 오브젝트(LOB) 위치 지정자 호스트 변수를 선언하는 구문을 나타냅니다.

C/C++에서 대형 오브젝트(LOB) 위치 지정자 호스트 변수의 구문





Variable

```
|--+---------------------------------------------------------------------+--|
   |  .-----------------.                                                |
   |  V                 |                                                |
   '----+-*-+--+----------+--variable-name--+------------------+---------'
        '-&-'  +-const----+                 '-= init-value-----'
               '-volatile-'
```

LOB 위치 지정자 호스트 변수 고려사항:

1. SQL TYPE IS, BLOB_LOCATOR, CLOB_LOCATOR, DBCLOB_LOCATOR는 대소문자 혼합으로 나타날 수 있습니다.
2. *init-value*는 포인터 및 참조 위치 지정자 변수의 초기화를 허용합니다. 다른 초기화 유형에는 아무 의미도 없습니다.

CLOB 위치 지정자 예(다른 LOB 위치 지정자 유형 선언은 비슷):

선언:

```
SQL TYPE IS CLOB_LOCATOR my_locator;
```

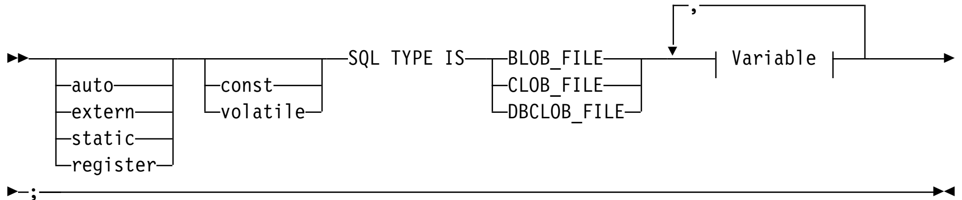
그 결과 다음 선언이 생성됩니다.

```
sqlint32 my_locator;
```

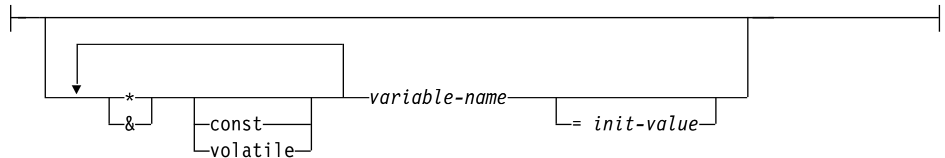
C 또는 C++의 파일 참조 선언

C/C++에서 파일 참조 호스트 변수의 구문은 C 또는 C++에서 파일 참조 호스트 변수를 선언하는 구문을 나타냅니다.

C/C++에서 파일 참조 호스트 변수의 구문



Variable



주:

- SQL TYPE IS, BLOB_FILE, CLOB_FILE, DBCLOB_FILE은 대소문자 혼합으로 나타날 수 있습니다.

CLOB 파일 참조 예(다른 LOB 파일 참조 선언은 비슷):

선언:

```
static volatile SQL TYPE IS BLOB_FILE my_file;
```

그 결과 다음 구조가 생성됩니다.

```
static volatile struct {
     sqluint32        name_length;
     sqluint32        data_length;
     sqluint32        file_options;
              char    name[255];
} my_file;
```

C 및 C++의 호스트 변수 초기화

C++ 선언 절에서는 괄호를 사용하여 호스트 변수를 초기화할 수 없습니다. 다음 예는 선언 절에서 올바르게 초기화하는 방법과 틀리게 초기화하는 방법을 나타냅니다.

```
EXEC SQL BEGIN DECLARE SECTION;
     short my_short_2 = 5;       /* correct   */
     short my_short_1(5);        /* incorrect */
EXEC SQL END DECLARE SECTION;
```

C 매크로 예

C/C++ 사전 처리 컴파일러는 선언 절에서 선언에 사용한 C 매크로를 직접 처리할 수 없습니다. 대신 외부 C 선행 처리기로 소스 파일을 먼저 선행 처리해야 합니다. 그렇게 하려면 PREPROCESSOR 옵션을 통해 사전 처리 컴파일러에 C 선행 처리기를 호출하는 정확한 명령을 지정하십시오.

PREPROCESSOR 옵션을 지정하면 사전 처리 컴파일러가 먼저 SQL INCLUDE 문에서 참조한 모든 파일의 내용을 소스 파일로 통합하여 모든 SQL INCLUDE 문을 처리합니다. 그런 다음 사전 처리 컴파일러에서 사용자가 수정한 소스 파일을 통해 입력으로 지정한 명령을 사용하여 외부 C 선행 처리기를 호출합니다. 선행 처리 파일(사전 처리 컴파일러에서 확장자를 항상 ".i"로 예상)은 사전 처리 컴파일 프로세스의 나머지에 대해 새로운 소스 파일로 사용됩니다.

사전 처리 컴파일러에서 생성한 #line 매크로는 더 이상 원래의 소스 파일을 참조하지 않는 대신 선행 처리 파일을 참조합니다. 컴파일러 오류를 원래의 소스 파일과 다시 관련시키려면 선행 처리 파일에 주석을 포함하십시오. 그렇게 하면 헤더 파일을 포함하여 원래 소스 파일의 다양한 절을 쉽게 찾을 수 있습니다. 주석을 포함한 옵션은 C 선행 처리기에서 공통적으로 사용할 수 있으며 PREPROCESSOR 옵션을 통해 지정한 명령에 이 옵션을 포함할 수 있습니다. 사전 처리 컴파일러에서 생성한 것과 틀리게 혼합될 수 있으므로 C 선행 처리기로 #line 매크로 자체를 출력해선 안됩니다.

매크로 확장 사용에 대한 주:

1. PREPROCESSOR 옵션을 통해 지정한 명령에 입력 파일의 이름을 제외한 모든 원하는 옵션을 포함해야 합니다. 예를 들어, AIX의 IBM C 경우 다음 옵션을 사용할 수 있습니다.

```
xlC -P -DMYMACRO=1
```

2. 사전 처리 컴파일러에서는 .i 확장자를 가진 선행 처리 파일을 생성하는 명령을 기대합니다. 그러나 재지정을 사용하여 선행 처리 파일을 생성할 수 없습니다. 예를 들어, 다음 옵션을 사용하여 선행 처리 파일을 생성할 수 **없습니다**.

   ```
   xlC -E > x.i
   ```

3. 외부 C 선행 처리기에서 발견하는 모든 오류는 원래 소스 파일에 해당하는 이름의 파일로 보고됩니다. 그러나 이 파일의 확장자는 되지만 .err입니다.

예를 들어, 다음과 같이 소스 코드에서 매크로 확장을 사용할 수 있습니다.

```
#define SIZE 3

  EXEC SQL BEGIN DECLARE SECTION;
  char a[SIZE+1];
  char b[(SIZE+1)*3];
        struct
  {
          short length;
    char  data[SIZE*6];
  } m;
  SQL TYPE IS BLOB(SIZE+1) x;
  SQL TYPE IS CLOB((SIZE+2)*3) y;
  SQL TYPE IS DBCLOB(SIZE*2K) z;
  EXEC SQL END DECLARE SECTION;
```

PREPROCESSOR 옵션 사용 후 위 선언은 다음으로 분석됩니다.

```
  EXEC SQL BEGIN DECLARE SECTION;
  char a[4];
  char b[12];
        struct
  {
          short length;
    char  data[18];
  } m;
  SQL TYPE IS BLOB(4) x;
  SQL TYPE IS CLOB(15) y;
  SQL TYPE IS DBCLOB(6144) z;
  EXEC SQL END DECLARE SECTION;
```

C 및 C++의 호스트 구조 지원

호스트 구조 지원을 통해 C/C++ 사전 처리 컴파일러에서 호스트 변수를 단일 호스트 구조로 그룹화할 수 있습니다. 따라서 SQL문의 동일 호스트 변수 집합을 빠

르게 참조할 수 있습니다. 예를 들어, 다음 호스트 구조를 사용하여 SAMPLE 데이터베이스의 STAFF 테이블에 있는 일부 컬럼에 액세스할 수 있습니다.

```
struct tag
   {
     short id;
     struct
     {
       short length;
       char  data[10];
     } name;
     struct
     {
       short   years;
       double salary;
     } info;
   } staff_record;
```

호스트 구조의 필드는 유효한 임의의 호스트 변수 유형이 될 수 있습니다. 여기에는 모든 숫자, 문자 및 대형 오브젝트(LOB) 유형이 포함됩니다. 중첩된 호스트 구조도 25 레벨까지 지원됩니다. 위의 예에서 `info` 필드는 하위 구조이지만 `name` 필드는 VARCHAR 필드를 표시하므로 하위 구조가 아닙니다. 같은 원칙이 LONG VARCHAR, VARGRAPHIC 및 LONG VARGRAPHIC에 적용됩니다. 호스트 구조의 포인터도 지원됩니다.

SQL문의 호스트 구조에서 그룹화된 호스트 변수를 참조하는 방법에는 두 가지가 있습니다.

1. 호스트 구조 이름은 SQL문에서 참조할 수 있습니다.

   ```
   EXEC SQL SELECT id, name, years, salary
        INTO :staff_record
             FROM staff
             WHERE id = 10;
   ```

   사전 처리 컴파일러에서 `staff_record`에 대한 참조를 호스트 구조에서 선언된 모든 필드의 목록으로 변환합니다. 이때 목록은 쉼표로 분리됩니다. 각 필드는 모든 레벨의 호스트 구조 이름으로 규정되므로 다른 호스트 변수나 필드와의 명명 충돌을 방지합니다. 이는 다음 방법과 일치합니다.

2. SQL문에서 완전한 호스트 변수 이름을 참조할 수 있습니다.

```
EXEC SQL SELECT id, name, years, salary
     INTO :staff_record.id, :staff_record.name,
          :staff_record.info.years, :staff_record.info.salary
          FROM staff
          WHERE id = 10;
```

같은 이름을 가진 다른 호스트 변수가 없는 경우에도 필드 이름에 대한 참조는 완전해야 합니다. 규정된 하위 구조도 참조할 수 있습니다. 위의 예에서 :staff_record.info를 사용하여 :staff_record.info.years, :staff_record.info.salary를 대체할 수 있습니다.

첫번째 예의 호스트 구조에 대한 참조가 쉼표로 분리되는 필드의 목록에 해당하므로 이런 유형의 참조가 오류를 일으킬 수 있는 인스턴스가 있습니다. 예를 들어, 다음과 같습니다.

```
EXEC SQL DELETE FROM :staff_record;
```

여기서 DELETE문은 단일 문자 기반 호스트 변수를 기대합니다. 호스트 구조를 제공하면 명령문에서 사전 처리 컴파일 시간 오류가 발생합니다.

```
SQL0087N  "staff_record" 호스트 변수는 구조 참조가 허용되지 않을 때
사용되는 구조입니다.
```

PREPARE, EXECUTE IMMEDIATE, CALL, 표시기 변수 및 SQLDA 참조 등 다른 호스트 구조를 사용하면 SQL0087N 오류가 발생할 수 있습니다. 두 번째 예에서 개별 필드를 정확히 참조할 때와 같은 상황에서는 한 필드만을 포함한 호스트 구조가 허용됩니다.

C 및 C++의 표시기 테이블

표시기 테이블은 호스트 구조에 사용할 표시기 변수의 콜렉션입니다. 이 변수 집합은 짧은 정수 배열로 선언해야 합니다. 예를 들면 다음과 같습니다.

```
short ind_tab[10];
```

위의 예는 10개의 구성요소가 있는 표시기 테이블을 선언합니다. 다음은 SQL문에서 이 테이블을 사용할 수 있는 방법을 나타냅니다.

```
EXEC SQL SELECT id, name, years, salary
     INTO :staff_record INDICATOR :ind_tab
          FROM staff
          WHERE id = 10;
```

다음에서는 테이블의 해당 표시기 변수를 포함한 각 호스트 구조 필드를 나열합니다.

staff_record.id	ind_tab[0]
staff_record.name	ind_tab[1]
staff_record.info.years	ind_tab[2]
staff_record.info.salary	ind_tab[3]

주: SQL문에서 `ind_tab[1]`과 같은 표시기 테이블 구성요소를 개별적으로 참조할 수 없습니다. INDICATOR 키워드는 선택적입니다. 구조 필드와 표시기의 수가 일치할 필요는 없습니다. 추가 표시기는 사용되지 않으며 추가 필드에서 표시기를 할당하지 않습니다.

표시기 변수 테이블 대신 스칼라 표시기 변수를 사용하여 호스트 구조의 첫번째 필드에 대한 표시기를 제공할 수도 있습니다. 이는 구성요소가 하나만 있는 표시기 테이블을 포함하는 것과 같습니다. 예를 들면 다음과 같습니다.

```
short scalar_ind;

EXEC SQL SELECT id, name, years, salary
          INTO :staff_record INDICATOR :scalar_ind
          FROM staff
          WHERE id = 10;
```

호스트 구조 대신 호스트 변수와 함께 표시기 테이블을 지정할 경우 `ind_tab[0]`과 같은 표시기 테이블의 첫번째 구성요소만 사용되지 않습니다.

```
EXEC SQL SELECT id
          INTO :staff_record.id INDICATOR :ind_tab
          FROM staff
          WHERE id = 10;
```

호스트 구조에서 짧은 정수 배열을 선언할 경우

```
struct tag
{
  short i[2];
} test_record;
```

`:test_record`를 `:test_record.i[0]`, `:test_record.i[1]`과 동일하게 설정하면서 SQL문에서 `test_record`를 참조하면 배열이 그 구성요소로 확장됩니다.

C 및 C++의 널로 종료된 문자열

C/C++ 널로 종료되는 문자열은 자체 SQLTYPE(문자의 경우 460/461 및 그래픽의 경우 468/469)를 포함합니다.

C/C++ 널로 종료되는 문자열은 LANGLEVEL 사전 처리 컴파일러 옵션의 값에 따라 다르게 조절됩니다. SQL문에서 이런 SQLTYPE 및 선언 길이 *n*에 해당하는 호스트 변수 중 하나를 지정하고, 데이터의 바이트 수(문자 유형의 경우) 또는 2 바이트 문자(그래픽 유형의 경우)가 *k*일 경우:

- PREP 명령에서 LANGLEVEL 옵션이 SAA1(기본값)이면

  출력의 경우:

조건...	결과...
k > *n*	*n* 문자가 목표 호스트 변수로 이동되고 SQLWARN1이 'W', SQLCODE 0 (SQLSTATE 01004)으로 설정됩니다. 널 종료기는 문자열에 지정되지 않습니다. 호스트 변수에 표시기 변수를 지정하면 표시기 변수의 값은 *k*로 설정됩니다.
k = *n*	*k* 문자가 목표 호스트 변수로 이동되고 SQLWARN1이 'N' 및 SQLCODE 0 (SQLSTATE 01004)으로 설정됩니다. 널 종료기는 문자열에 지정되지 않습니다. 호스트 변수에 표시기 변수를 지정하면 표시기 변수의 값은 0으로 설정됩니다.
k < *n*	*k* 문자가 목표 호스트 변수로 이동되고 널 문자가 *k* + *1* 문자에 지정됩니다. 호스트 변수에 표시기 변수를 지정하면 표시기 변수의 값은 0으로 설정됩니다.

  입력의 경우: 데이터베이스 관리 프로그램에서 널 종료기로 끝나지 않는 SQLTYPE 중 한 입력 호스트 변수를 발견할 경우 *n+1* 문자에 널 종료기 문자가 포함된다고 가정합니다.

- PREP 명령에서 LANGLEVEL 옵션이 MIA이면

출력의 경우:

조건...	결과...
$k >= n$	$n - 1$ 문자가 목표 호스트 변수로 이동되고 SQLWARN1이 'W' 및 SQLCODE 0 (SQLSTATE 01501)으로 설정됩니다. *n*번째 문자가 널 종료기로 설정됩니다. 호스트 변수에 표시기 변수를 지정하면 표시기 변수의 값은 *k*로 설정됩니다.
$k + 1 = n$	*k* 문자가 목표 호스트 변수로 이동되고 널 종료기가 *n* 문자에 지정됩니다. 호스트 변수에 표시기 변수를 지정하면 표시기 변수의 값은 0으로 설정됩니다.
$k + 1 < n$	*k* 문자가 목표 호스트 변수로 이동되고 $k + 1$ 문자의 오른쪽 시작 부분에 $n - k -1$개의 공백이 추가되고 널 종료기가 *n* 문자에 지정됩니다. 호스트 변수에 표시기 변수를 지정하면 표시기 변수의 값은 0으로 설정됩니다.

입력의 경우: 데이터베이스 관리 프로그램에서 널 종료기로 끝나지 않는 SQLTYPE 중 한 입력 호스트 변수를 발견할 경우 -302 (SQLSTATE 22501)가 리턴됩니다.

다른 SQL 문맥에서 지정할 경우 길이가 *n*인 호스트 변수 SQLTYPE 460은 위에서 정의한 *n* 길이의 VARCHAR 데이터 유형으로 처리됩니다. 다른 SQL 문맥에서 지정할 경우 길이가 *n*인 호스트 변수 SQLTYPE 468은 위에서 정의한 *n* 길이의 VARGRAPHIC 데이터 유형으로 처리됩니다.

C 및 C++의 포인터 데이터 유형

다음 제한사항 아래서 특정 데이터 유형의 포인터로 호스트 변수를 선언할 수 있습니다.

- 호스트 변수를 포인터로 선언할 경우 동일 소스 파일에서 동일 이름을 가진 다른 호스트 변수는 선언할 수 없습니다. 다음 예는 허용되지 않습니다.

```
char mystring[20];
char (*mystring)[20];
```

- 널로 종료되는 문자 배열의 포인터를 선언할 경우에는 괄호를 사용하십시오. 다른 모든 경우에는 괄호를 사용할 수 없습니다. 예를 들면 다음과 같습니다.

  ```
  EXEC SQL BEGIN DECLARE SECTION;
       char  (*arr)[10];  /* correct   */
       char  *(arr);      /* incorrect */
       char  *arr[10];    /* incorrect */
  EXEC SQL END DECLARE SECTION;
  ```

  첫번째 선언은 10 바이트 문자 배열의 포인터입니다. 이는 유효한 호스트 변수입니다. 두 번째는 유효하지 않은 선언입니다. 문자의 포인터에서 괄호는 허용되지 않습니다. 세 번째 선언은 포인터의 배열입니다. 이는 지원되지 않는 데이터 유형입니다.

  호스트 변수 선언:

  ```
  char *ptr
  ```

  위의 선언은 수용되지만 *결정되지 않은 길이의 널로 종료되는 문자열*을 의미하지는 않습니다. 대신 *고정 길이의, 단일 문자 호스트 변수의 포인터*를 의미합니다. 이는 의도한 내용이 아닐 수도 있습니다. 다른 문자열을 지시할 수 있는 포인터 호스트 변수를 정의하려면 위의 첫번째 선언 양식을 사용하십시오.

- SQL문에서 포인터 호스트 변수를 사용할 경우 다음 예와 같이 선언할 때 사용한 것과 같은 수의 별표를 접두부에 지정해야 합니다.

  ```
  EXEC SQL BEGIN DECLARE SECTION;
    char (*mychar)[20];    /* Pointer to character array of 20 bytes */
  EXEC SQL END DECLARE SECTION;

  EXEC SQL SELECT column INTO :*mychar FROM table;   /* Correct */
  ```

- 호스트 변수 이름 위에 별표만을 연산자로 사용할 수 있습니다.
- 별표는 이름의 일부로 간주되지 않으므로 지정한 별표의 수가 호스트 변수 이름의 최대 길이에 영향을 주지 않습니다.
- SQL문에서 포인터 변수를 사용할 때마다 최적화 레벨 사전 처리 컴파일 옵션(OPTLEVEL)을 기본 설정값 0(최적화 안함)으로 두어야 합니다. 즉, 데이터베이스 관리 프로그램에서 어떤 SQLDA 최적화도 실행하지 않습니다.

C 및 C++에서 클래스 데이터 구성원을 호스트 변수로 사용

클래스나 오브젝트 자체가 아닌 호스트 변수로 클래스 데이터 구성원을 선언할 수 있습니다. 다음 예는 사용할 메소드를 나타냅니다.

```
    class STAFF
    {
        private:
  EXEC SQL BEGIN DECLARE SECTION;
               char         staff_name[20];
               short int    staff_id;
               double       staff_salary;
  EXEC SQL END DECLARE SECTION;
             short        staff_in_db;
        .
        .
    };
```

클래스 구성원 함수에서 C++ 컴파일러가 제공하는 내재된 *this* 포인터를 통해서만 SQL문에서 데이터 구성원에 액세스할 수 있습니다. SQL문에서 SELECT name INTO :my_obj.staff_name ...과 같은 오브젝트 인스턴스를 명시적으로 규정할 수 **없습니다**.

SQL문에서 클래스 데이터 구성원을 직접 참조할 경우 데이터베이스 관리 프로그램에서 *this* 포인터를 사용하여 참조를 분석합니다. 이 때문에 최적화 레벨 사전 처리 컴파일 옵션(OPTLEVEL)을 기본 설정값 0(최적화 안함)으로 두어야 합니다. 즉, 데이터베이스 관리 프로그램에서 어떤 SQLDA 최적화도 실행하지 않습니다. 이는 SQL문에 포인터 호스트 변수가 포함될 때마다 적용됩니다.

다음 예는 SQL문에서 호스트 변수로 선언한 클래스 데이터 구성원을 직접 사용하는 방법을 나타냅니다.

```
    class STAFF
    {
⋮
        public:

⋮
           short int hire( void )
```

```
        {
          EXEC SQL INSERT INTO staff ( name,id,salary )
            VALUES ( :staff_name, :staff_id, :staff_salary );
          staff_in_db = (sqlca.sqlcode == 0);
          return sqlca.sqlcode;
        }
    };
```

이 예에서는 클래스 데이터 구성원 staff_name, staff_id 및 staff_salary을 INSERT문에서 직접 사용합니다. 이런 구성원은 호스트 변수로 선언되므로(692 페이지의 『호스트 변수로 클래스 데이터 구성원 선언 예』의 예 참조), *this* 포인터를 통해 현재 오브젝트에 내재적으로 규정됩니다. SQL문에서 *this* 포인터를 통해 액세스할 수 없는 데이터 구성원을 참조할 수도 있습니다. 포인터 또는 참조 호스트 변수를 사용하여 간접적으로 참조해서 그렇게 할 수 있습니다.

다음 예는 두 번째 오브젝트 *otherGuy*를 취하는 새로운 메소드 *asWellPaidAs*를 나타냅니다. SQL문에서 그 구성원을 직접적으로 참조할 수 없으므로, 이 메소드는 지역 포인터나 참조 호스트 변수를 통해 그 구성원을 간접적으로 참조합니다.

```
     short int STAFF::asWellPaidAs( STAFF otherGuy )
     {
  EXEC SQL BEGIN DECLARE SECTION;
           short &otherID = otherGuy.staff_id
           double otherSalary;
  EXEC SQL END DECLARE SECTION;
         EXEC SQL SELECT SALARY INTO :otherSalary
           FROM STAFF WHERE id = :otherID;
           if( sqlca.sqlcode == 0 )
              return staff_salary >= otherSalary;
           else
              return 0;
     }
```

C 및 C++에서 구성 및 구성원 연산자 사용

Embedded SQL문에서 C/C++ 구성원 연산자 '.' 또는 '->'나 C++ 범위 분석 연산자 '::'를 사용할 수 **없습니다**. SQL문 외부에서 원하는 범위 변수를 지시하도록 설정한 다음 SQL문 안에서 이를 지시하는 데 사용하는 지역 포인터나 참조 변수를 통해 같은 작업을 쉽게 할 수 있습니다. 다음 예는 사용할 올바른 메소드를 나타냅니다.

```
EXEC SQL BEGIN DECLARE SECTION;
     char (& localName)[20] = ::name;
EXEC SQL END DECLARE SECTION;
   EXEC SQL
     SELECT name INTO :localName FROM STAFF
     WHERE name = 'Sanders';
```

C 및 C++에서의 그래픽 호스트 변수 처리

C 또는 C++ 응용프로그램에서 그래픽 데이터를 조절하려면 DB2에서 제공하는 sqldbchar 데이터 유형이나 wchar_t C/C++ 데이터 유형에 따라 호스트 변수를 사용하십시오. 이런 호스트 변수의 유형을 GRAPHIC, VARGRAPHIC 또는 DBCLOB인 테이블 컬럼에 할당할 수 있습니다. 예를 들어, 테이블의 GRAPHIC 또는 VARGRAPHIC 컬럼에서 DBCS 데이터를 갱신하거나 선택할 수 있습니다.

그래픽 호스트 변수의 유효한 양식은 다음과 같이 세 가지입니다.

- 단일 그래픽 양식

  단일 그래픽 호스트 변수에는 GRAPHIC(1) SQL 데이터 유형에 해당하는 468/469의 SQLTYPE가 있습니다. 676 페이지의 그래픽 선언 구문(단일 그래픽 양식과 널로 종료되는 그래픽 양식)에서 자세한 내용을 참조하십시오.

- 널로 종료되는 그래픽 양식

  널로 종료된다는 것은, 그래픽 문자열 중 마지막 문자의 모든 바이트에 2진 0('\0's)이 포함되는 상황을 의미합니다. 여기에는 400/401의 SQLTYPE가 포함됩니다. 676 페이지의 그래픽 선언 구문(단일 그래픽 양식과 널로 종료되는 그래픽 양식)에서 자세한 내용을 참조하십시오.

- VARGRAPHIC 구조 양식

  그 길이가 1과 16 336 사이일 경우 VARGRAPHIC 구조 호스트 변수에는 464/465의 SQLTYPE가 포함됩니다. 그 길이가 2000과 16 350 바이트 사이일 경우 472/473의 SQLTYPE가 포함됩니다. 678 페이지의 그래픽 선언의 구문(VARGRAPHIC 구조 양식)에서 자세한 내용을 참조하십시오.

C 및 C++의 다중 바이트 문자 인코딩

일부 문자 엔코딩 스키마, 특히 서아시아 국가의 문자 엔코딩 스키마의 경우 문자를 표시하는 데 복수 바이트가 필요합니다. 데이터에 대한 외부 표시를 문자에 대

한 *복수 바이트 문자 코드* 표시라 하며 여기에는 2 바이트 문자(2 바이트로 표시되는 문자)가 포함됩니다. DB2의 그래픽 데이터는 2 바이트 문자로 구성됩니다.

2 바이트 문자로 문자열을 조작하려면 응용프로그램에서 데이터에 대한 내부 표시를 사용하는 것이 편리할 수 있습니다. 이 내부 표시를 2 바이트 문자에 대한 *와이드 문자 코드*라 하며 이 표시는 `wchar_t` C/C++ 데이터 유형에 관습적으로 사용되는 형식입니다. ANSI C 및 X/OPEN Portability Guide 4(XPG4)를 따르는 서브루틴을 사용하여 와이드 문자 데이터를 처리하고 와이드 문자 형식의 데이터와 복수 바이트 형식 사이에서 변환할 수 있습니다.

응용프로그램에서 복수 바이트 형식이나 와이드 문자 형식으로 문자를 처리할 수 있지만 DBCS(복수 바이트) 문자 코드만으로 데이터베이스 관리 프로그램과의 상호작용이 실행된다는 점에 주의하십시오. 즉, 데이터는 GRAPHIC 컬럼에 DBCS 형식으로 저장되며 여기서 검색됩니다. WCHARTYPE 사전 처리 컴파일러 옵션은, 데이터베이스 엔진과 교환할 때 와이드 문자 형식의 응용프로그램 데이터를 복수 바이트 형식으로 또는 그 반대로 변환할 수 있도록 제공됩니다.

C 및 C++에서 wchar_t 또는 sqldbchar 데이터 유형 선택

특정 코드 페이지에 대해 DB2 그래픽 데이터의 *크기*와 엔코딩이 플랫폼 사이에서 일정하지만 ANSI C 또는 C++ `wchar_t` 데이터 유형의 *크기*와 내부 형식은 사용하는 컴파일러와 사용자가 있는 플랫폼에 따라 달라집니다. 그러나 DB2에서 `sqldbchar` 데이터 유형을 2 바이트 *크기*로 정의하며, 이는 데이터베이스에 정의된 것과 같은 형식으로 DBCS 및 UCS-2 데이터를 조작하는 일종의 이식 방식입니다. UCS-2 데이터에 대해서는 581 페이지의『일본어 및 대만어 EUC 및 UCS-2 코드 세트 변환』과 *관리 안내서*에서 자세한 내용을 참조하십시오.

`wchar_t` 또는 `sqldbchar`을 사용하여 모든 DB2 C 그래픽 호스트 변수 유형을 정의할 수 있습니다. 696 페이지의『C 및 C++의 WCHARTYPE 사전 처리 컴파일러 옵션』에서 설명한 대로 WCHARTYPE CONVERT 사전 처리 컴파일 옵션을 사용하여 응용프로그램을 빌드할 경우 `wchar_t`를 사용해야 합니다.

주: Windows 플랫폼에서 WCHARTYPE CONVERT 옵션을 지정할 경우 Windows 플랫폼의 wchar_t가 유니코드라는 점에 주의해야 합니다. 따라서 C/C++ 컴파일러의 wchar_t가 유니코드가 아닐 경우 wcstombs() 유니코드 함수 호출이 SQLCODE -1421 (SQLSTATE=22504)로 실패할 수 있습니다.

이런 경우 WCHARTYPE NOCONVERT 옵션을 지정하고, 프로그램에서 wcstombs() 및 mbstowcs() 함수를 명시적으로 호출할 수 있습니다.

WCHARTYPE NOCONVERT 사전 처리 컴파일 옵션으로 응용프로그램을 빌드할 경우 서로 다른 DB2 클라이언트와 서버 플랫폼간의 최대 이식성을 위해 sqldbchar을 사용해야 합니다. WCHARTYPE NOCONVERT에 wchar_t를 사용할 수 있지만 wchar_t를 2 바이트 길이로 정의한 플랫폼에서만 가능합니다.

호스트 변수 선언에서 wchar_t나 sqldbchar을 틀리게 사용할 경우 사전 처리 컴파일시 SQLCODE 15 (no SQLSTATE)를 수신하게 됩니다.

C 및 C++의 WCHARTYPE 사전 처리 컴파일러 옵션

WCHARTYPE 사전 처리 컴파일러 옵션을 사용하여, C/C++ 응용프로그램에서 사용하려는 그래픽 문자를 지정할 수 있습니다. 이 옵션을 사용하면 그래픽 데이터를 복수 바이트 형식으로 유지할 지 아니면 와이드 문자 형식으로 유지할 지 융통성있게 선택할 수 있습니다. WCHARTYPE 옵션의 가능한 값은 다음과 같이 두 가지입니다.

CONVERT

WCHARTYPE CONVERT 옵션을 선택하면 그래픽 호스트 변수와 데이터베이스 관리 프로그램 사이에서 문자 코드가 변환됩니다. 그래픽 입력 호스트 변수의 경우 ANSI C 함수 wcstombs()를 사용하여 데이터를 데이터베이스 관리 프로그램으로 전송하기 전에 와이드 문자 형식에서 복수 바이트 DBCS 문자 형식으로의 문자 코드 변환이 수행됩니다. 그래픽 출력 호스트 변수의 경우 ANSI C 함수 mbstowcs()를 사용하여 호스트 변수에 저장된 데이터베이스 관리 프로그램에서 데이터를 수신하기 전에 복수 바이트 DBCS 문자 형식에서 와이드 문자 형식으로의 문자 코드 변환이 수행됩니다.

WCHARTYPE CONVERT를 사용할 경우의 이점은, 데이터베이스 관리 프로그램과 통신하기 전에 명시적으로 데이터를 복수 바이트 형식으로 변환하지 않고도 응용프로그램에서 L-리터럴이나 'wc' 문자열 함수와 같은 와이드 문자열 처리를 위해 ANSI C 메카니즘을 완전하게 활용할 수 있다는 것입니다. 단점은, 내재된 변환이 런타임에서 응용프로그램의 성능에 영향을 줄 수 있으며 메모리 요구사항을 증가시킬 수 있다는 것입니다.

WCHARTYPE CONVERT를 선택할 경우 sqldbchar 대신 wchar_t를 사용하여 모든 그래픽 호스트 변수를 선언하십시오.

WCHARTYPE CONVERT 수행을 원하지만 CLI 응용프로그램과 같은 사용자의 응용프로그램을 사전 처리 컴파일할 필요가 없을 경우 컴파일 타임에서 C 선행 처리기 매크로 SQL_WCHART_CONVERT를 정의하십시오. 그렇게 하면 DB2 헤더 파일의 특정 정의에서 sqldbchar 대신 wchar_t 데이터 유형을 사용하게 됩니다.

주: DB2 Windows 3.1 클라이언트에서 수행 중인 프로그램에서는 현재 WCHARTYPE CONVERT 사전 처리 컴파일 옵션을 지원하지 않습니다. 이런 프로그램의 경우 기본값(WCHARTYPE NOCONVERT)을 사용하십시오.

NOCONVERT (기본값)

WCHARTYPE NOCONVERT 옵션을 선택하거나 WCHARTYPE 옵션을 지정하지 않을 경우 응용프로그램과 데이터베이스 관리 프로그램 사이에 내재적 문자 코드 변환은 발생하지 않습니다. 그래픽 호스트 변수의 데이터는 변경되지 않은 DBCS 문자로 데이터베이스 관리 프로그램로 전송되거나 반대로 수신됩니다. 이 경우 성능이 향상된다는 이점이 있지만 응용프로그램에서 wchar_t 호스트 변수에 와이드 문자 데이터를 사용하지 않거나 데이터베이스 관리 프로그램과 상호작용할 때 명시적으로 wcstombs() 및 mbstowcs() 함수를 호출하여 데이터와 복수 바이트 형식 사이에서 변환해야 한다는 단점이 있습니다.

WCHARTYPE NOCONVERT를 선택할 경우 다른 DB2 클라이언트/서버 플랫폼과의 이식성을 최대화하기 위해 sqldbchar 유형을 사용하여 모든 그래픽 호스트 변수를 선언하십시오.

*Command Reference*에서 자세한 내용을 참조하십시오.

다음은 사용자가 따라야 할 기타 지침입니다.

- wchar_t 또는 sqldbchar 지원을 사용하여 DBCS 데이터를 조절하므로 이를 사용하려면 DBCS 또는 EUC 지원 하드웨어 및 소프트웨어가 필요합니다. 이 지원은 DB2 Universal Database의 DBCS 환경에서만 사용하거나 1 바이트 응

용프로그램을 포함하여 UCS-2 데이터베이스에 연결된 모든 응용프로그램의 GRAPHIC 데이터를 처리하기 위해서만 사용할 수 있습니다.

- 그래픽 문자열에는 비-DBCS 문자와, 비-DBCS 문자로 변환할 수 있는 와이드 문자를 사용하지 않아야 합니다. *비-DBCS 문자*는 1 바이트 문자와 2 바이트 이외의 문자를 의미합니다. 그래픽 문자열로는 그 값에 2 바이트 문자 코드 포인트만 포함되었는지 확인할 수 없습니다. 그래픽 호스트 변수에는 DBCS 데이터만 포함되거나 WCHARTYPE CONVERT가 적용된 경우 DBCS 데이터로 변환될 와이드 문자 데이터만 포함되어야 합니다. 문자 호스트 변수에는 2 바이트와 1 바이트가 혼합된 데이터를 저장합니다. 혼합 데이터 호스트 변수는 WCHARTYPE 옵션 설정의 영향을 받지 않습니다.
- WCHARTYPE NOCONVERT 사전 처리 컴파일 옵션을 사용한 응용프로그램에서 L-리터럴은 와이드 문자 형식이므로 L-리터럴은 그래픽 호스트 변수와 결합하여 사용해선 안됩니다. L-리터럴은 `"wchar_t의 배열"` 데이터 유형의 문자 L이 접두부에 지정된 C 와이드 문자열 리터럴입니다. 예를 들면, `L"dbcs-문자열이` L-리터럴입니다.
- WCHARTYPE CONVERT 사전 처리 컴파일 옵션을 사용한 응용프로그램에서 L-리터럴을 사용하여 `wchar_t` 호스트 변수를 초기화할 수 있지만 SQL문에서는 사용할 수 없습니다. L-리터럴을 사용하는 대신 SQL문은 그래픽 문자열 상수를 사용합니다. 이 상수는 WCHARTYPE 설정에 독립적입니다.
- WCHARTYPE 옵션 설정은 호스트 변수는 물론 SQLDA 구조를 사용하여 데이터베이스 관리 프로그램와 주고받는 그래픽 데이터에 영향을 줍니다. WCHARTYPE CONVERT를 적용할 경우 SQLDA를 통해 응용프로그램에서 수신한 그래픽 데이터는 와이드 문자 형식을 취하며 `wcstombs()`에 대한 내재적 호출을 통해 DBCS 형식으로 변환될 것이라 가정됩니다. 마찬가지로 응용프로그램에서 수신한 그래픽 출력 데이터는 응용프로그램 저장영역에 지정되기 전에 와이드 문자로 변환됩니다.
- 비분리 저장 프로시듀어는 WCHARTYPE NOCONVERT 옵션으로 사전 처리 컴파일해야 합니다. 일반 분리 저장 프로시듀어는 CONVERT 또는 NOCONVERT 옵션으로 사전 처리 컴파일할 수 있습니다. 이 옵션은 저장 프로시듀어에 포함된 SQL문으로 조작하는 그래픽 데이터의 형식에 영향을 줍니다. 그러나 어떤 경우든 SQLDA를 통해 저장 프로시듀어에 전달되는 모든 그

래픽 데이터의 형식은 DBCS 형식이어야 합니다. 마찬가지로 SQLDA를 통해 저장 프로시듀어에서 가져오는 데이터의 형식도 DBCS 형식이어야 합니다.

- 응용프로그램에서 DARI(Database Application Remote Interface) (sqleproc() API)를 통해 저장 프로시듀어를 호출할 경우, 응용프로그램의 WCHARTYPE 설정 호출 상태에 상관없이 입력 SQLDA의 모든 그래픽 데이터의 형식은 DBCS 형식이거나 UCS-2 데이터베이스에 연결된 경우 UCS-2 형식이어야 합니다. 마찬가지로 SQLDA 출력의 모든 그래픽 데이터는 WCHARTYPE 설정에 상관없이 DBCS 형식으로 리턴되거나 UCS-2 데이터베이스에 연결된 경우 UCS-2 형식으로 리턴됩니다.
- 응용프로그램에서 SQL CALL문을 통해 저장 프로시듀어를 호출할 경우 응용프로그램의 WCHARTYPE 설정 호출에 따라 그래픽 데이터 변환이 SQLDA에서 발생합니다.
- 사용자 정의 함수(UDF)로 전달된 그래픽 데이터의 형식은 항상 DBCS 형식입니다. 마찬가지로 UDF에서 리턴되는 모든 그래픽 데이터의 형식은 DBCS 데이터베이스의 경우 DBCS 형식이고, EUC 및 UCS-2 데이터베이스의 경우 UCS-2라 가정합니다.
- DBCLOB 파일 참조 변수를 통해 DBCLOB 파일에 저장한 데이터는 DBCS 형식이나 UCS-2 데이터베이스의 경우 UCS-2 형식으로 저장됩니다. 마찬가지로 DBCLOB 파일의 입력 데이터는 DBCS 형식이나 UCS-2 데이터베이스의 경우 UCS-2 형식으로 검색합니다.

주:

1. WCHARTYPE CONVERT 옵션을 사용하여 C 응용프로그램을 사전 처리 컴파일할 경우 DB2에서 변환 함수를 통해 데이터를 전달하는 입력 및 출력시 응용프로그램의 그래픽 데이터가 유효한 지 확인합니다. CONVERT 옵션을 사용하지 **않을** 경우 그래픽 데이터의 변환이 발생하지 않으므로 유효성 검증은 발생하지 않습니다. 혼합 CONVERT/NOCONVERT 환경에서, NOCONVERT 응용프로그램이 유효하지 않은 그래픽 데이터를 삽입한 다음 CONVERT 응용프로그램이 분리할 경우에는 문제가 발생할 수 있습니다. 이 데이터는 CONVERT 응용프로그램의 FETCH에서 SQLCODE -1421(SQLSTATE 22504)를 사용하여 변환하지 못합니다.

2. DB2 Windows 3.1 클라이언트에서 수행 중인 프로그램의 경우 현재 WCHARTYPE CONVERT 사전 처리 컴파일 옵션은 지원되지 않습니다. 이 경우, 기본 WCHARTYPE NOCONVERT 옵션을 사용하십시오.

C 및 C++의 일본어 또는 대만어 EUC 및 UCS-2 고려사항

응용프로그램 코드 페이지가 일보언 또는 대만어 EUC이거나 응용프로그램에서 UCS-2 데이터베이스에 연결할 경우 CONVERT 또는 NOCONVERT 옵션, `wchar_t` 또는 `sqldbchar` 그래픽 호스트 변수 또는 입력/출력 SQLDA를 사용하여 데이터베이스 서버에 있는 GRAPHIC 컬럼에 액세스할 수 있습니다. 이 절에서 *DBCS* 형식은 EUC 데이터에 대한 UCS-2 엔코딩 스키마를 의미합니다. 다음 경우를 고려하십시오.

- CONVERT 옵션 사용

  DB2 클라이언트가 와이드 문자 형식에서 응용프로그램 코드 페이지로 데이터를 변환한 다음 입력 SQLDA를 데이터베이스 서버로 전송하기 전에 UCS-2로 변환합니다. 데이터베이스 서버로 전송되는 모든 그래픽 데이터에는 UCS-2 코드 페이지 식별자가 태그로 지정됩니다. 혼합 문자 데이터에는 응용프로그램 코드 페이지 식별자가 태그로 지정됩니다. 클라이언트가 데이터베이스에서 그래픽 데이터를 검색할 때 데이터에 UCS-2 코드 페이지 식별자가 태그로 지정됩니다. DB2 클라이언트가 UCS-2에서 클라이언트 응용프로그램 코드 페이지로 변환한 다음 와이드 문자 형식으로 변환합니다. 호스트 변수 대신 입력 SQLDA를 사용할 경우 와이드 문자 형식을 사용하여 그래픽 데이터를 엔코딩했는지 확인해야 합니다. 이 데이터는 UCS-2로 변환된 다음 데이터베이스 서버로 전송됩니다. 이런 변환은 성능에 영향을 줍니다.

- NOCONVERT 옵션 사용

  그래픽 데이터는 DB2에서 UCS-2를 사용하여 엔코딩하고 UCS-2 코드 페이지를 통해 태그가 지정되며 변환은 실행되지 않습니다. DB2에서는 그래픽 호스트 변수가 단지 한 집합으로 사용되고 있다고 가정합니다. NOCONVERT 옵션을 선택하면 데이터베이스 서버에서 검색한 그래픽 데이터가 UCS-2를 사용하여 엔코딩한 응용프로그램으로 전달됩니다. 응용프로그램 코드 페이지에서 UCS-2로, UCS-2에서 응용프로그램 코드 페이지로의 모든 변환은 사용자가 담당해야 합니다. UCS-2 식별자 태그가 지정된 데이터는 변환이나 변경없이 데이터베이스 서버로 전송됩니다.

변환을 최소화하기 위해 NOCONVERT 옵션을 사용하고 응용프로그램에서 변환을 조절하거나 GRAPHIC 컬럼을 사용하지 않을 수 있습니다. Windows NT 또는 AIX 버전 4.3 이상과 같이 wchar_t 엔코딩이 2 바이트 유니코드인 클라이언트 환경의 경우 NOCONVERT 옵션을 사용하여 UCS-2에 대해 직접 작업할 수 있습니다. 이런 경우 사용자의 응용프로그램에서 큰 끝 형식과 작은 끝 형식 사이의 차이점을 조절해야 합니다. NOCONVERT 옵션을 사용할 경우 DB2 Universal Database에서 항상 2 바이트 큰 끝 형식인 sqldbchar을 사용합니다.

NOCONVERT를 지정한 경우 UCS-2으로 변환 후 또는 CONVERT를 지정한 경우 와이드 문자 형식으로 변환하여 IBM-eucJP/IBM-eucTW CS0 (7 비트 ASCII) 및 IBM-eucJP CS2 (Katakana) 데이터를 그래픽 호스트 변수에 할당하지 마십시오. UCS-2에서 PC DBCS로 변환할 때 이런 EUC 코드 세트에 있는 문자가 1 바이트가 되기 때문입니다.

일반적으로 eucJP 및 eucTW에서 GRAPHIC 데이터를 UCS-2로 저장하더라도 이런 데이터베이스의 GRAPHIC 데이터는 여전히 비-ASCII eucJP 또는 eucTW 데이터입니다. 특히 이런 GRAPHIC 데이터에 채워지는 공백은 DBCS 공백(UCS-2, U+3000의 표의적인 공백이라고도 함)입니다. 그러나 UCS-2 데이터베이스의 경우 GRAPHIC 데이터에는 모든 UCS-2 문자가 들어갈 수 있으며 공백 채움은 UCS-2 공백, U+0020으로 실행됩니다. UCS-2 데이터베이스에서 UCS-2 데이터 대 eucJP 및 eucTW 데이터베이스에서 UCS-2 데이터를 검색하도록 응용프로그램을 코딩할 때 이 차이점을 기억해 두어야 합니다.

일반 EUC 응용프로그램 개발 지침에 대해서는 581 페이지의 『일본어 및 대만어 EUC 및 UCS-2 코드 세트 변환』에서 자세한 내용을 참조하십시오.

C 및 C++의 지원되는 SQL 데이터 유형

사전 정의한 특정 C 및 C++ 데이터 유형은 데이터베이스 관리 프로그램 컬럼 유형에 해당합니다. 이런 C/C++ 데이터 유형만 호스트 변수로 선언할 수 있습니다.

702 페이지의 표30는 각 컬럼 유형에 해당하는 C/C++ 데이터 유형을 나타냅니다. 사전 처리 컴파일러에서 호스트 변수 선언을 찾으면 해당 SQL 유형 값을 판별합니다. 데이터베이스 관리 프로그램에서 이 값을 사용하여 응용프로그램과 자체에서 교환되는 데이터를 변환합니다.

주: 모든 DB2 호스트 언어에서 DATALINK 데이터 유형에 대한 호스트 변수 지원은 없습니다.

표 30. C/C++ 선언에 맵핑되는 SQL 데이터 유형

SQL 컬럼 유형[1]	**C/C++ 데이터 유형**	**SQL 컬럼 유형 설명**
SMALLINT (500 또는 501)	short short int sqlint16	부호있는 16비트 정수
INTEGER (496 또는 497)	long long int sqlint32[2]	부호있는 32비트 정수
BIGINT (492 또는 493)	long long long __int64 sqlint64[3]	부호있는 64비트 정수
REAL[4] (480 또는 481)	float	단정밀 부동 소수점
DOUBLE[5] (480 또는 481)	double	배정밀 부동 소수점
DECIMAL(*p*,*s*) (484 또는 485)	정확한 해당 데이터 유형 없음, double 사용	팩된 10진수 CHAR 및 DECIMAL 함수를 사용하여 압축 십진수 필드를 문자 데이터로 조작할 것을 고려하십시오.
CHAR(1) (452 또는 453)	char	단일 문자
CHAR(*n*) (452 또는 453)	정확한 해당 데이터 유형 없음, char[*n*+*1*] 사용. 여기서 n은 데이터를 유지할만큼 충분히 큽니다. 1<=*n*<=254	고정 길이 문자열

VARCHAR(*n*)
(448 또는 449)
	struct tag {
	short int;
	char[*n*]
	}
	
	1<=*n*<=32 672
		2바이트 문자열 길이 표시기를 포함하며 널로 종료되는 가변 문자열은 없습니다.

	또는 char[*n*+*1*] 사용. 여기서 n은 데이터를 유지할 만큼 충분히 큽니다. 1<=*n*<=32 672	널로 종료되는 변수 길이 문자열 **주:** 할당된 460/461의 SQL 유형

표 30. C/C++ 선언에 맵핑되는 SQL 데이터 유형 (계속)

SQL 컬럼 유형[1]	C/C++ 데이터 유형	SQL 컬럼 유형 설명

LONG VARCHAR
(456 또는 457)
	struct tag {
	short int;
	char[*n*]
	}
	
	32 673<=*n*<=32 700
		2바이트 문자열 길이 표시기를 포함하며 널로 종료되는 가변 문자열은 없습니다.

CLOB(*n*) (408 또는 409)	sql type is clob(*n*) 1<=*n*<=2 147 483 647	4 바이트 문자열 길이 표시기를 포함하며 널로 종료되는 가변 문자열은 없습니다.
CLOB 위치 지정자 변수[6] (964 또는 965)	sql type is clob_locator	서버의 CLOB 엔터티를 식별합니다.
CLOB 파일 참조 변수[6] (920 또는 921)	sql type is clob_file	CLOB 데이터를 포함한 파일의 설명자
BLOB(*n*) (404 또는 405)	sql type is blob(*n*) 1<=*n*<=2 147 483 647	4 바이트 문자열 길이 표시기를 포함하며 널로 종료되는 가변 2진 문자열은 없습니다.
BLOB 위치 지정자 변수[6] (960 또는 961)	sql type is blob_locator	서버의 BLOB 엔터티를 식별합니다.
BLOB 파일 참조 변수[6] (916 또는 917)	sql type is blob_file	BLOB 데이터를 포함한 파일의 설명자
DATE (384 또는 385)	널로 종료되는 문자 양식	널 종료기를 수용하도록 적어도 11 문자를 허용합니다.
	VARCHAR 구조 양식	적어도 10 문자를 허용합니다.
TIME (388 또는 389)	널로 종료되는 문자 양식	널 종료기를 수용하도록 적어도 9 문자를 허용합니다.
	VARCHAR 구조 양식	적어도 8 문자를 허용합니다.
TIMESTAMP (392 또는 393)	널로 종료되는 문자 양식	널 종료기를 수용하도록 적어도 27 문자를 허용합니다.
	VARCHAR 구조 양식	적어도 26 문자를 허용합니다.
주: WCHARTYPE NOCONVERT 옵션으로 사전 처리 컴파일할 때 DBCS 또는 EUC 환경에서만 다음 데이터 유형을 사용할 수 있습니다.		
GRAPHIC(1) (468 또는 469)	sqldbchar	단일 2 바이트 문자
GRAPHIC(*n*) (468 또는 469)	정확한 해당 데이터 유형 없음, sqldbchar[*n+1*] 사용. 여기서 n은 데이터를 유지할만큼 충분히 큽니다. 1<=*n*<=127	고정 길이 2 바이트 문자열

표 30. C/C++ 선언에 맵핑되는 SQL 데이터 유형 (계속)

SQL 컬럼 유형[1]	C/C++ 데이터 유형	SQL 컬럼 유형 설명

VARGRAPHIC(*n*)
(464 또는 465)
	struct tag {
	short int;
	sqldbchar[*n*]
	}
	
	1<=*n*<=16 336
		2 바이트 문자열 길이 표시기를 포함하며 널로 종료되는 가변 2 바이트 문자열은 없습니다.

	또는 sqldbchar[*n+1*] 사용. 여기서 n은 데이터를 유지할만큼 충분히 큽니다. 1<=*n*<=16 336	널로 종료되는 변수 길이 2 바이트 문자열 **주:** 할당된 400/401의 SQL 유형

LONG VARGRAPHIC
(472 또는 473)
	struct tag {
	short int;
	sqldbchar[*n*]
	}
	
	16 337<=*n*<=16 350
		2 바이트 문자열 길이 표시기를 포함하며 널로 종료되는 가변 2 바이트 문자열은 없습니다.

주: WCHARTYPE CONVERT 옵션으로 사전 처리 컴파일할 때 DBCS 또는 EUC 환경에서만 다음 데이터 유형을 사용할 수 있습니다.		
GRAPHIC(1) (468 또는 469)	wchar_t	• C-유형의 경우 단일 와이드 문자 • 컬럼 유형의 경우 단일 2 바이트 문자
GRAPHIC(*n*) (468 또는 469)	정확한 해당 데이터 유형 없음, wchar_t [*n+1*] 사용. 여기서 n은 데이터를 유지할만큼 충분히 큽니다. 1<=*n*<=127	고정 길이 2 바이트 문자열

VARGRAPHIC(*n*)
(464 또는 465)
	struct tag {
	short int;
	wchar_t [*n*]
	}
	
	1<=*n*<=16 336
		2 바이트 문자열 길이 표시기를 포함하며 널로 종료되는 가변 2 바이트 문자열은 없습니다.

	또는 char[*n+1*] 사용. 여기서 n은 데이터를 유지할만큼 충분히 큽니다. 1<=*n*<=16 336	널로 종료되는 변수 길이 2 바이트 문자열 **주:** 할당된 400/401의 SQL 유형

LONG VARGRAPHIC
(472 또는 473)
	struct tag {
	short int;
	wchar_t [*n*]
	}
	
	16 337<=*n*<=16 350
		2 바이트 문자열 길이 표시기를 포함하며 널로 종료되는 가변 2 바이트 문자열은 없습니다.

표 30. C/C++ 선언에 맵핑되는 SQL 데이터 유형 (계속)

SQL 컬럼 유형[1]	C/C++ 데이터 유형	SQL 컬럼 유형 설명
주: DBCS 또는 EUC 환경에서만 다음 데이터 유형을 사용할 수 있습니다.		
DBCLOB(*n*) (412 또는 413)	sql type is dbclob(*n*) 1<=*n*<=1 073 741 823	4 바이트 문자열 길이 표시기를 포함하며 널로 종료되는 가변 2 바이트 문자열은 없습니다.
DBCLOB 위치 지정자 변수[6] (968 또는 969)	sql type is dbclob_locator	서버의 DBCLOB 엔터티를 식별합니다.
DBCLOB 파일 참조 변수[6] (924 또는 925)	sql type is dbclob_file	DBCLOB 데이터를 포함한 파일의 설명자

주:

1. **SQL 컬럼 유형** 아래서 첫번째 숫자는 표시기 변수가 제공되지 않은 경우를 나타내며 두 번째 숫자는 표시기 변수가 제공된 경우를 나타냅니다. 표시기 변수는 널(NULL) 값을 나타내거나 절단된 문자열의 길이를 유지하는 데 필요합니다. 이들은 해당 데이터 유형에 대해 SQLDA의 SQLTYPE에 나타나는 값입니다.
2. 플랫폼 호환성을 위해 sqlint32를 사용하십시오. 64-비트 UNIX 플랫폼에서 ″긴″은 64비트 정수입니다. 64-비트 Windows 운영 체제와 32-비트 UNIX 플랫폼에서 ″긴″은 32비트 정수입니다.
3. 플랫폼 호환성을 위해서는 sqlint64를 사용하십시오. Microsoft 컴파일러를 사용할 경우 DB2 Universal Database sqlsystm.h 헤더 파일은 Windows NT 플랫폼에서 sqlint64를 ″__int64″로 유형 정의합니다. 32비트 UNIX 플랫폼에서는 ″long long″으로 정의하고 64비트 UNIX 플랫폼에서는 ″long″으로 유형 정의됩니다.
4. FLOAT(*n*) 여기서 $0 < n < 25$는 REAL의 동의어입니다. SQLDA에서 REAL과 DOUBLE 사이의 차이점은 길이 값(4 또는 8)입니다.
5. 다음 SQL 유형은 DOUBLE의 동의어입니다.
   - FLOAT
   - FLOAT(*n*) 여기서 $24 < n < 54$
   - DOUBLE PRECISION
6. 이는 컬럼 유형이 아닌 호스트 변수 유형입니다.

다음은 지원되는 SQL 데이터 유형에 대해 선언된 호스트 변수를 포함한 샘플 SQL 선언 절입니다.

```
EXEC SQL BEGIN DECLARE SECTION;

⋮

      short     age = 26;                /* SQL type  500 */
      short     year;                    /* SQL type  500 */
      sqlint32  salary;                  /* SQL type  496 */
```

```
sqlint32  deptno;                 /* SQL type  496 */
float     bonus;                  /* SQL type  480 */
double    wage;                   /* SQL type  480 */
char      mi;                     /* SQL type  452 */
char      name[6];                /* SQL type  460 */
struct   {
          short len;
          char data[24];
         } address;               /* SQL type  448 */
struct   {
          short len;
          char data[32695];
         } voice;                 /* SQL type  456 */
sql type is clob(1m)
          chapter;                /* SQL type  408 */
sql type is clob_locator
          chapter_locator;        /* SQL type  964 */
sql type is clob_file
          chapter_file_ref;       /* SQL type  920 */
sql type is blob(1m)
          video;                  /* SQL type  404 */
sql type is blob_locator
          video_locator;          /* SQL type  960 */
sql type is blob_file
          video_file_ref;         /* SQL type  916 */
sql type is dbclob(1m)
          tokyo_phone_dir;        /* SQL type  412 */
sql type is dbclob_locator
          tokyo_phone_dir_lctr;   /* SQL type  968 */
sql type is dbclob_file
          tokyo_phone_dir_flref;  /* SQL type  924 */
struct   {
          short len;
          sqldbchar data[100];
         } vargraphic1;           /* SQL type  464 */
                                  /* Precompiled with
                                  WCHARTYPE NOCONVERT option */
struct   {
          short len;
          wchar_t data[100];
         } vargraphic2;           /* SQL type  464 */
                                  /* Precompiled with
                                  WCHARTYPE CONVERT option */
struct   {
          short len;
          sqldbchar data[10000];
         } long_vargraphic1;      /* SQL type  472 */
                                  /* Precompiled with
```

```
                                              WCHARTYPE NOCONVERT option */
      struct    {
                 short len;
                 wchar_t data[10000];
                } long_vargraphic2;           /* SQL type  472 */
                                              /* Precompiled with
                                              WCHARTYPE CONVERT option */
      sqldbchar graphic1[100];                /* SQL type  468 */
                                              /* Precompiled with
                                              WCHARTYPE NOCONVERT option */
      wchar_t   graphic2[100];                /* SQL type  468 */
                                              /* Precompiled with
                                              WCHARTYPE CONVERT option */
      char      date[11];                     /* SQL type  384 */
      char      time[9];                      /* SQL type  388 */
      char      timestamp[27];                /* SQL type  392 */
      short     wage_ind;                     /* Null indicator */

.
.
.
  EXEC SQL END DECLARE SECTION;
```

다음은 지원되는 C/C++ 데이터 유형에 대한 추가 규칙입니다.

- char 데이터 유형은 char 또는 unsigned char로 선언할 수 있습니다.
- 데이터베이스 관리 프로그램에서 널로 종료되는 변수 길이 문자열 데이터 유형 char[*n*](데이터 유형 460)을 VARCHAR(*m*)로 처리합니다.
  - LANGLEVEL이 SAA1이면 호스트 변수 길이 *m*은 char[*n*]의 문자열 길이 *n* 또는 첫번째 널 종료기(\0) 앞에 나오는 바이트 수 중 작은 쪽과 동일합니다.
  - LANGLEVEL이 MIA이면 호스트 변수 길이 *m*은 첫번째 널 종료기(\0) 앞에 나오는 바이트 수와 동일합니다.
- 데이터베이스 관리 프로그램에서는 널로 종료되는 변수 길이 그래픽 문자열 데이터 유형 wchar_t[*n*] 또는 sqldbchar[*n*](데이터 유형 400)을 VARGRAPHIC(*m*)로 처리합니다.
  - LANGLEVEL이 SAA1이면 호스트 변수 변수 길이 *m*은 wchar_t[*n*] 또는 sqldbchar[*n*]의 문자열 길이 *n*이나 첫번째 그래픽 널 종료기 앞에 나오는 문자 수 중 작은 쪽과 동일합니다.

- LANGLEVEL이 MIA이면 호스트 변수 변수 길이 *m*은 첫번째 그래픽 널 종료기 앞에 나오는 문자 수와 동일합니다.

- 부호없는 숫자 데이터 유형은 지원하지 않습니다.
- 내부 표현이 머신에 따라 달라지므로 C/C++ 데이터 유형 int는 허용되지 않습니다.

C 및 C++의 2진 데이터용

표준 C 또는 C++ 문자열 유형 460은 2진 데이터용으로 지정된 컬럼에 사용해선 안됩니다. 널 문자가 나타나면 데이터베이스 관리 프로그램에서 이 데이터 유형을 절단합니다. VARCHAR(SQL 유형 448) 또는 CLOB(SQL 유형 408) 구조를 사용하십시오.

C 및 C++의 SQLSTATE 및 SQLCODE 변수

SQL92E 값에 LANGLEVEL 사전 처리 컴파일 옵션을 사용할 경우 다음 두 선언이 호스트 변수로 포함될 수 있습니다.

```
EXEC SQL BEGIN DECLARE SECTION;
  char  SQLSTATE[6]
  sqlint32  SQLCODE;

  .
  .
  .

EXEC SQL END DECLARE SECTION;
```

이들 중 어느 것도 지정하지 않을 경우 사전 처리 컴파일 단계 중 SQLCODE 선언이 취해집니다. 이 옵션을 사용할 때 INCLUDE SQLCA문을 지정해선 안됩니다. 이 점에 주의하십시오.

복수 소스 파일로 구성된 응용프로그램에서, SQLCODE 및 SQLSTATE 변수는 첫번째 소스 파일에서 위와 같이 정의될 수 있습니다. 후속 소스 파일에서는 다음과 같이 정의를 수정해야 합니다.

```
extern sqlint32 SQLCODE;
extern char SQLSTATE[6];
```

제21장 Java 프로그래밍

Java에 대한 프로그래밍 고려사항 709
SQLJ와 JDBC 비교. 710
다른 언어를 능가하는 Java의 이점 . . . 710
Java에 있는 SQL 보안. 711
Java에 대한 소스 및 출력 파일 711
Java 클래스 라이브러리. 711
Java 패키지 712
Java에서 지원되는 SQL 데이터 유형 . . 712
Java에 있는 SQLSTATE 및 SQLCODE 값 714
Java 추적 기능 715
Java 응용프로그램 및 애플릿 작성 . . . 715
작업 방식 716
JDBC 프로그래밍. 718
DB2Appl 프로그램의 작업 방식. . . . 719
JDBC 예: DB2Appl.java 721
JDBC 응용프로그램 분배 722
JDBC 애플릿 분배 및 수행 722
JDBC 2.0 723
JDBC 2.0 코어 API 지원. 723
JDBC 2.0 선택적 패키지 API 지원 724
JDBC 2.0 호환성 726
SQLJ 프로그래밍. 726
DB2 SQLJ 지원 727
DB2 SQLJ 제한사항 728
Java에 SQL문 포함 730
SQLJ에서의 반복기 동작 선언 . . . 730
SQLJ 예: App.sqlj 732
Java 호스트 변수. 736
SQLJ에 있는 저장 프로시듀어 및 함수에 대한 호출 736
SQLJ 프로그램 컴파일 및 수행 737
SQLJ 변환기 옵션 739
Java에 있는 저장 프로시듀어 및 UDF . . 740
Java 클래스를 넣을 위치 741
루틴에서 Java 클래스 갱신 742
Java에서 저장 프로시듀어 디버깅 . . . 743
디버그 준비. 743
디버그 테이블 데이터 상주. 744
디버거 호출. 746
Java 저장 프로시듀어 및 UDF 746
JAR 파일 설치, 대체 및 제거 . . . 747
Java 함수 정의 749
JDBC 1.22에 LOB 및 그래픽 오브젝트 사용 750
JDBC 및 SQLJ 상호운용성 752
세션 공유 752
Java의 연결 자원 관리 752

Java에 대한 프로그래밍 고려사항

DB2 Universal Database에서는 두 표준 Java 프로그래밍 즉, JDBC(Java Database Connectivity) 및 SQLJ(Embedded SQL for Java)를 구현합니다. 이 장에서는 JDBC 및 SQLJ 프로그래밍에 대해 개요이지만, DB2 고유의 측면에 초점을 맞춥니다. JDBC 및 SQLJ 스펙의 링크에 대해서는 `http://www.ibm.com/software/data/db2/java/`의 DB2 Universal Database Java 웹 사이트를 참조하십시오.

© Copyright IBM Corp. 1993, 2000

SQLJ와 JDBC 비교

JDBC API를 통해 데이터베이스에 대한 동적 SQL 호출을 작성하는 Java 프로그램을 작성할 수 있습니다. SQLJ 응용프로그램에서는 데이터베이스에 연결하여 SQL 오류를 조절하는 타스크의 기초로 JDBC를 사용하지만 SQLJ 소스 파일에 정적 Embedded SQL문도 포함할 수 있습니다. 결과 Java 소스 파일을 컴파일하려면 먼저 SQLJ 변환프로그램으로 SQLJ 소스 파일을 변환해야 합니다.

JDBC 및 SQLJ 응용프로그램 빌드에 대해서는 *응용프로그램 빌드 안내서*에서 자세한 내용을 참조하십시오.

다른 언어를 능가하는 Java의 이점

Embedded SQL을 포함하는 프로그래밍 언어를 호스트 언어라고 합니다. Java는 SQL을 Ebed하는 방법에 상당한 영향을 주는 방식에서 전형적인 호스트 언어 C, COBOL 및 FORTRAN과 다릅니다.

- SQLJ 및 JDBC는 개방형 표준이므로, 다른 표준 호환 데이터베이스 시스템에서 DB2 Universal Database로 SQLJ 또는 JDBC 응용프로그램을 쉽게 이식할 수 있습니다.
- 혼합 데이터와 다양한 크기의 데이터를 나타내는 모든 Java 유형은 구별 값 `null`을 가지며, 이 값은 SQL NULL 상태를 표시하는 데 사용될 수 있으므로, Java 프로그램에 다른 호스트 언어의 널(NULL) 표시기를 대체할 수 있는 수단을 제공합니다.
- Java는 자동으로 이질 이식이 가능한("super portable" 또는 단지 "다운로드 가능"이라고도 함) 프로그램을 지원하도록 설계되었습니다. Java 유형 시스템 클래스 및 인터페이스와 함께, 이 기능은 구성요소 소프트웨어를 작동 가능하게 합니다. 특히, Java로 작성된 SQLJ 변환 프로그램은 데이터베이스 벤더가 권한 부여, 스키마 검사, 유형 검사, 트랜잭션 및 복구 기능과 같은 기존 데이터베이스 기능을 향상시키고 특정 데이터베이스에 최적화된 코드를 생성하기 위해 전문화한 구성요소를 호출할 수 있습니다.
- Java는 이질 네트워크에서 2진 이식이 가능하도록 설계되며, 이로써 정적 SQL을 사용하는 데이터베이스 응용프로그램의 2진 이식이 가능해집니다.

Java에 있는 SQL 보안

기본적으로 JDBC 프로그램은 프로그램을 수행하는 사람에게 할당된 특권으로 SQL 문을 실행합니다. 반대로 SQLJ 프로그램은 데이터베이스 패키지를 작성한 사람에게 할당된 특권으로 SQL문을 실행합니다.

Java에 대한 소스 및 출력 파일

소스 파일의 확장자는 다음과 같습니다.

.java Java 소스 파일, 사전 처리 컴파일을 실행할 필요가 없습니다. 사용자의 Java 개발 환경에 포함된 `javac` Java 컴파일러로 이런 파일을 컴파일할 수 있습니다.

.sqlj SQLJ 소스 파일, `sqlj` 변환 프로그램으로 변환해야 합니다. 변환 프로그램에서는 다음을 작성합니다.

- 하나 이상의 `.class` 바이트코드 파일
- 연결 문맥 당 한 `.ser` 프로파일 파일

해당 출력 파일의 확장자는 다음과 같습니다.

.class JDBC 및 SQLJ 바이트코드 컴파일 파일

.ser SQLJ 직렬화된 프로파일 파일. 데이터베이스에서 `db2profc` 유틸리티로 각 프로파일 파일에 대해 패키지를 작성합니다.

SQLJ 프로그램을 컴파일해서 수행하는 방법의 예는 737 페이지의 『SQLJ 프로그램 컴파일 및 수행』에서 볼 수 있습니다.

Java 클래스 라이브러리

DB2 Universal Database에서는 JDBC 및 SQLJ 지원용 클래스 라이브러리를 제공합니다. 이런 라이브러리는 CLASSPATH에 제공하거나 다음과 같이 애플릿으로 포함해야 합니다.

db2java.zip

저장 프로시듀어와 UDF 지원을 포함하여 JDBC 및 SQLJ 지원 클래스를 제공합니다.

sqlj.zip

SQLJ 변환 프로그램 클래스 파일을 제공합니다.

runtime.zip
: SQLJ 응용프로그램 및 애플릿에 대한 Java 런타임 지원을 제공합니다.

Java 패키지

사용자의 응용프로그램에서 DB2에 포함된 클래스 라이브러리를 사용하려면 소스 파일의 맨 위에 해당 `import` *package*문을 포함해야 합니다. Java 응용프로그램에서 다음 패키지를 사용할 수 있습니다.

java.sql.*
: JDK에 포함된 JDBC API. 모든 JDBC 및 SQLJ 프로그램에서 이 패키지를 가져와야 합니다.

sqlj.runtime.*
: 모든 DB2 클라이언트에 대하여 SQLJ를 지원합니다. 모든 SQLJ 프로그램에서 이 패키지를 가져와야 합니다.

sqlj.runtime.ref.*
: 모든 DB2 클라이언트에 대하여 SQLJ를 지원합니다. 모든 SQLJ 프로그램에서 이 패키지를 가져와야 합니다.

Java에서 지원되는 SQL 데이터 유형

713 페이지의 표31은 데이터 유형 맵핑에 대한 JDBC 스펙에 기초하여 각 SQL 데이터 유형에 해당하는 Java 데이터 유형을 나타냅니다. 일부 맵핑은 JDBC 버전 1.22 또는 버전 2.0 드라이버 사용 여부에 따라 달라집니다. 이 점에 주의하십시오. JDBC 드라이버는 다음 맵핑 스키마를 사용하여 응용프로그램과 데이터베이스 사이에서 교환되는 데이터를 변환합니다. 사용자의 Java 응용프로그램과 PARAMETER STYLE JAVA 저장 프로시듀어 및 UDF에서 해당 맵핑을 사용하십시오. PARAMETER STYLE DB2GENERAL 저장 프로시듀어와 UDF의 데이터 유형 맵핑에 대해서는 863 페이지의 『지원되는 SQL 데이터 유형』의 내용을 참조하십시오.

주: DB2에서 지원하는 모든 프로그래밍 언어에서 DATALINK 데이터 유형에 대한 호스트 변수 지원은 없습니다.

표 31. Java 선언에 맵핑되는 SQL 데이터 유형

SQL 컬럼 유형	**Java 데이터 유형**	**SQL 컬럼 유형 설명**
SMALLINT (500 또는 501)	short	부호있는 16비트 정수
INTEGER (496 또는 497)	int	부호있는 32비트 정수
BIGINT (492 또는 493)	long	부호있는 64비트 정수
REAL (480 또는 481)	float	단정밀 부동 소수점
DOUBLE (480 또는 481)	double	배정밀 부동 소수점
DECIMAL(*p*,*s*) (484 또는 485)	java.math.BigDecimal	팩된 10진수
CHAR(*n*) (452 또는 453)	String	*n* 길이의 고정 길이 문자열. 여기서 *n*의 범위는 1 - 254입니다.
VARCHAR(*n*) (448 또는 449)	String	가변 길이 문자열
LONG VARCHAR (456 또는 457)	String	긴 가변 길이 문자열
CHAR(*n*) FOR BIT DATA	byte[]	*n* 길이의 고정 길이 문자열. 여기서 *n*의 범위는 1 - 254입니다.
VARCHAR(*n*) FOR BIT DATA	byte[]	가변 길이 문자열
LONG VARCHAR FOR BIT DATA	byte[]	긴 가변 길이 문자열
BLOB(*n*) (404 또는 405)	JDBC 1.22: byte[] JDBC 2.0: java.sql.Blob	대형 오브젝트(LOB) 가변 길이 2진 문자열
CLOB(*n*) (408 또는 409)	JDBC 1.22: String JDBC 2.0: java.sql.Clob	대형 오브젝트(LOB) 가변 길이 문자열
DBCLOB(*n*) (412 또는 413)	JDBC 1.22: String JDBC 2.0: java.sql.Clob	대형 오브젝트(LOB) 가변 길이 2 바이트 문자열
DATE (384 또는 385)	java.sql.Date	10바이트 문자열
TIME (388 또는 389)	java.sql.Time	8바이트 문자열

표 31. Java 선언에 맵핑되는 SQL 데이터 유형 (계속)

SQL 컬럼 유형	Java 데이터 유형	SQL 컬럼 유형 설명
TIMESTAMP (392 또는 393)	java.sql.Timestamp	26바이트 문자열

Java에 있는 SQLSTATE 및 SQLCODE 값

SQL 오류가 발생하면 JDBC와 SQLJ 프로그램은 SQLException을 발생시킵니다. SQLException에서 SQLSTATE, SQLCODE 또는 SQLMSG 값을 검색하려면 다음과 같이 해당 인스턴스 메소드를 호출하십시오.

SQL 리턴 코드	SQLException 메소드
SQLCODE	SQLException.getErrorCode()
SQLMSG	SQLException.getMessage()
SQLSTATE	SQLException.getSQLState()

예를 들어 다음과 같습니다.

```
int sqlCode=0;      // Variable to hold SQLCODE
String sqlState="00000";  // Variable to hold SQLSTATE

try
{
  // JDBC statements may throw SQLExceptions
  stmt.executeQuery("Your JDBC statement here");

  // SQLJ statements may also throw SQLExeptions
  #sql {..... your SQLJ statement here ......};
}

/* Here's how you can check for SQLCODEs and SQLSTATE */

catch (SQLException e)
{
  sqlCode = e.getErrorCode() // Get SQLCODE
  sqlState = e.getSQLState() // Get SQLSTATE

  if (sqlCode == -190 || sqlState.equals("42837"))
  {
    // Your code here to handle SQLCODE -190 or SQLSTATE 42837
  }
  else
```

```
      {
        // Your code here to handle other errors
      }
      System.err.println(e.getMessage()) ; // Print the exception
      System.exit(1);         // Exit
    }
```

Java 추적 기능

CLI/ODBC/JDBC 추적 기능과 DB2 추적 기능 db2trc를 사용하여 JDBC 또는 SQLJ 프로그램에 관련된 문제를 진단할 수 있습니다. 위의 추적을 실행하는 방법은 *문제점 해결 안내서*에 설명되어 있습니다.

SQLJ 프로그램에 런타임 호출 추적 기능을 설치할 수 있습니다. 유틸리티는 프로그램과 연관된 프로파일을 작동합니다. 프로그램에서 App_SJProfile0이라 하는 프로파일을 사용한다고 가정해봅시다. 프로그램에 호출 추적을 설치하려면 다음 명령을 사용하십시오.

```
profdb App_SJProfile0.ser
```

profdb 유틸리티는 JVM을 사용하여 sqlj.runtime.profile.util.AuditorInstaller 클래스의 main() 메소드를 수행합니다. AuditorInstaller 클래스 사용 및 옵션에 대해서는 http://www.ibm.com/software/data/db2/java 의 DB2 Java 웹 사이트를 참조하십시오.

Java 응용프로그램 및 애플릿 작성

응용프로그램 또는 애플릿에서 JDBC 또는 SQLJ를 사용하는 지에 따라 JDBC 스펙은 Sun Microsystems에서 얻을 수 있습니다. JDBC 및 SQLJ 자원의 링크에 대해서는 http://www.ibm.com/software/data/db2/java/의 DB2 Java 웹 사이트를 참조하십시오. 이 스펙에서는 JDBC API를 호출하여 데이터베이스에 액세스한 다음 그 데이터베이스의 데이터를 조작하는 방법에 대해 설명합니다.

또한 이 절을 읽어 JDBC에 대한 DB2의 확장과 그 제한사항에 대해 알아야 합니다. 723 페이지의 『JDBC 2.0』에서 자세한 내용을 참조하십시오. Java로 UDF나 저장 프로시듀어를 작성하려는 경우 466 페이지의 『Java 사용자 정의 함수 작

성 및 사용』 및 746 페이지의『Java 저장 프로시듀어 및 UDF』에서 자세한 내용을 참조하십시오. 여기서는 다른 언어보다 Java의 경우 달라지는 고려사항을 다룹니다.

JDBC 및 SQLJ 응용프로그램 및 애플릿을 빌드하려면 *응용프로그램 빌드 안내서*의 지침에 따라 사용자의 운영 체제 환경을 설정해야 합니다.

작업 방식

DB2의 Java 지원에는 세 가지 독립 구성요소가 있습니다.

- JDBC를 사용하여 DB2에 액세스하는, Java로 작성한 클라이언트 응용프로그램 및 애플릿에 대한 지원(718 페이지의『JDBC 프로그래밍』 참조)
- SQLJ를 사용하여 DB2에 액세스하는, Java로 작성한 클라이언트 응용프로그램 및 애플릿에 대한 사전 처리 컴파일 및 바인딩 지원(726 페이지의『SQLJ 프로그래밍』 참조)
- 서버의 Java UDF 및 저장 프로시듀어에 대한 지원(740 페이지의『Java에 있는 저장 프로시듀어 및 UDF』 참조)

Java 응용프로그램 지원: 717 페이지의 그림21은 DB2 JDBC 응용프로그램의 작업 방식을 나타냅니다. DB2 JDBC 응용프로그램을 DB2 CLI 응용프로그램으로 간주할 수 있으며 Java 언어를 사용하여 작성합니다. Java 원시 메소드를 통해 JDBC에 대한 호출은 DB2 CLI에 대한 호출로 변환됩니다. JDBC에서는 DB2 클라이언트로부터 DB2 CLI를 통해 DB2 서버에 이르는 흐름을 요청합니다.

SQLJ 응용프로그램은 이 JDBC 지원을 사용하며, 사전 처리 컴파일 및 바인딩 단계에서 데이터베이스에 바인드된 모든 SQL 패키지를 인증하고 실행하도록 SQLJ 런타임 클래스에 요구합니다.

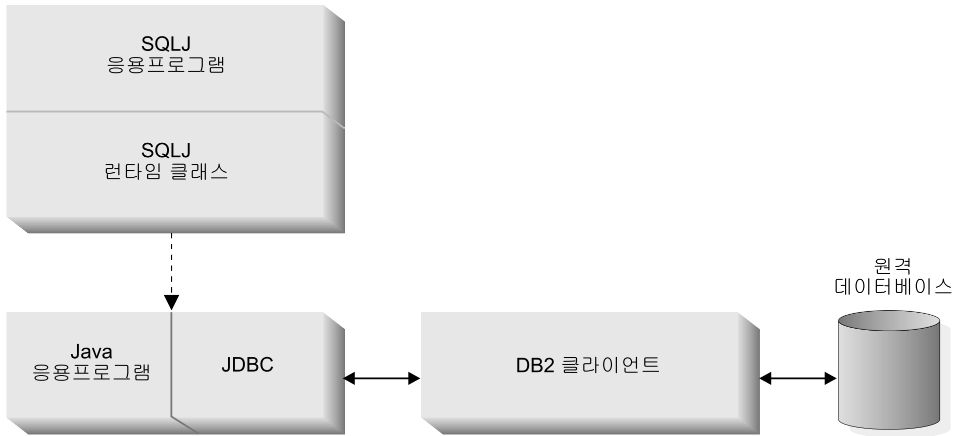


그림 21. DB2의 Java 응용프로그램 구현

Java 애플릿 지원: 718 페이지의 그림22은 *네트 드라이버*라는 JDBC *애플릿 드라이버*의 작업 방식을 설명합니다. 드라이버는 JDBC 클라이언트와 JDBC 서버 db2jd로 구성됩니다. JDBC 클라이언트 드라이버는 애플릿과 함께 웹 브라우저에 로드됩니다. 애플릿에서 DB2 데이터베이스에 대한 연결을 요청하면 클라이언트에서 웹 서버가 수행 중인 머신의 JDBC 서버에 대한 TCP/IP 소켓을 엽니다. 연결을 설정하면 클라이언트가 TCP/IP 연결을 통해 애플릿에서 JDBC 서버로의 후속 데이터베이스 액세스 요청을 전송합니다. 그런 다음 JDBC 서버에서 해당 CLI(ODBC)를 호출하여 타스크를 수행합니다. 완료시 JDBC 서버에서 연결을 통해 클라이언트로 결과를 다시 전송합니다.

SQLJ 애플릿에서 JDBC 클라이언트 드라이버 위에 SQLJ 클라이언트 드라이버를 추가하지만 그렇지 않은 경우 JDBC 애플릿과 같은 작업을 합니다.

DB2 JDBC 서버 시작에 대한 내용은 *Command Reference*의 db2jstrt 명령을 참조하십시오.

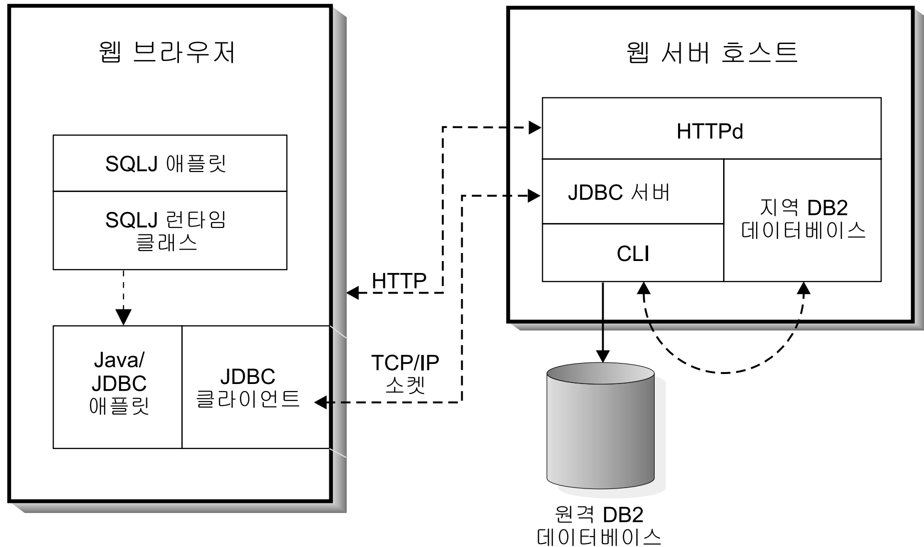


그림 22. DB2 Java 애플릿 구현

JDBC 프로그래밍

응용프로그램과 애플릿 둘다 다음과 같은 작업을 수행합니다.

1. 해당 Java 패키지와 클래스(`java.sql.*`)를 가져오십시오.
2. 해당 JDBC 드라이버(응용프로그램의 경우 `COM.ibm.db2.jdbc.app.DB2Driver` 또는 애플릿의 경우 `COM.ibm.db2.jdbc.net.DB2Driver`)를 로드하십시오.
3. JDBC 스펙에서 정의한 URL로 위치를 지정하고 `db2` 서브프로토콜을 사용하여 데이터베이스에 연결하십시오. 애플릿의 경우 사용자 ID, 암호, 호스트 이름 및 애플릿 서버의 포트 번호를 제공해야 합니다. 응용프로그램은 대체 값을 명시적으로 지정하지 않는 이상 DB2 클라이언트 카탈로그에서 사용자 ID와 암호의 기본값을 암시적으로 사용합니다.
4. 데이터베이스로 SQL문을 전달하십시오.
5. 결과를 수신하십시오.
6. 연결을 닫으십시오.

프로그램 코딩 후 다른 Java 프로그램의 경우와 마찬가지로 컴파일하십시오. 특수한 사전 처리 컴파일 또는 바인드 단계를 수행할 필요가 없습니다.

DB2Appl 프로그램의 작업 방식

다음 샘플 프로그램 DB2Appl.java는 DB2용 JDBC 프로그램 코딩 방법의 데모를 나타냅니다.

1. **JDBC 패키지 가져오기.** 모든 JDBC 및 SQLJ 프로그램에서는 JDBC 패키지를 가져와야 합니다.
2. **연결 오브젝트 선언.** 연결 오브젝트에서 데이터베이스 연결을 설정하고 관리합니다.
3. **데이터베이스 URL 변수를 설정합니다.** DB2 응용프로그램 드라이버에서는 `jdbc:db2:>`*`database name`*`<` 양식을 취하는 URL을 수용합니다.
4. **데이터베이스에 연결.** `DriverManager.getConnection()` 메소드는 보통 다음 매개변수와 함께 사용됩니다.

   `getConnection(String` *`url`***`)`**
   : 기본 사용자 ID와 암호로 *url*에 지정된 데이터베이스에 대한 연결을 설정합니다.

   `getConnection(String` *`url`***`, String`** *`userid`***`, String`** *`password`***`)`**
   : 각기 *userid*와 *passwd*에 지정된 사용자 ID와 암호로 *url*에 지정된 데이터베이스에 대한 연결을 설정합니다.

5. **명령문 오브젝트 작성.** 명령문 오브젝트는 데이터베이스로 SQL문을 전송합니다.
6. **SQL SELECT문을 실행합니다.** SELECT문과 같이 단일 결과 세트를 리턴하는 SQL문의 `executeQuery()` 메소드를 사용합니다. ResultSet 오브젝트에 결과를 할당합니다.
7. **ResultSet의 행 검색.** ResultSet 오브젝트를 사용하여 호스트 언어 Embedded SQL에서 커서와 같은 결과 세트를 처리할 수 있습니다. `ResultSet.next()` 메소드는다음 행으로 커서를 이동하고, 결과 세트의 마지막 행에 도달한 경우 `boolean false` 값을 리턴합니다. 결과 세트 처리에 대한 제한사항은 데이터베이스 관리 프로그램 구성 매개변수를 통해 사용되는 JDBC API의 레벨에 따라 달라집니다.

- JDBC 2.0 API를 통해 행을 결과 세트의 정방향이나 역방향으로 화면 이동할 수 있습니다.
- JDBC 1.22 API는 `ResultSet.next()` 메소드를 사용한 결과 세트의 정방향 화면 이동을 제한합니다.

8. **컬럼의 값 리턴.** `ResultSet.getString(`*n*`)`은 n^{th} 컬럼의 값을 문자열 오브젝트로 리턴합니다.
9. **SQL UPDATE문 실행.** SQL UPDATE문의 `executeUpdate()` 메소드를 사용합니다. 이 메소드는 갱신된 행의 수를 정수 값으로 리턴합니다.

JDBC 예: DB2Appl.java

```
import java.sql.*; 1

class DB2Appl {

   static {
      try {
         Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
      } catch (Exception e) {
         System.out.println(e);
      }
   }

   public static void main(String argv[]) {
      Connection con = null; 2

      // URL is jdbc:db2:dbname
      String url = "jdbc:db2:sample"; 3

      try {
         if (argv.length == 0) {
            // connect with default id/password
            con = DriverManager.getConnection(url);
            }
         else if (argv.length == 2) {
            String userid = argv[0];
            String passwd = argv[1];

            // connect with user-provided username and password
            con = DriverManager.getConnection(url, userid, passwd); 4
            }
         else {
            System.out.println("Usage: java DB2Appl [username password]");
            System.exit(0);
         }

         // retrieve data from the database
         System.out.println("Retrieve some data from the database...");
         Statement stmt = con.createStatement(); 5
         ResultSet rs = stmt.executeQuery("SELECT * from employee"); 6

         System.out.println("Received results:");

         // display the result set
         // rs.next() returns false when there are no more rows
         while (rs.next()) { 7
            String a = rs.getString(1); 8
            String str = rs.getString(2);

            System.out.print(" empno= " + a);
            System.out.print(" firstname= " + str);
            System.out.print("");
         }

         rs.close();
         stmt.close();

         // update the database
         System.out.println("Update the database... ");
         stmt = con.createStatement();
         int rowsUpdated = stmt.executeUpdate("UPDATE employee
```

```
         SET firstnme = 'SHILI' where empno = '000010'"); 9

          System.out.print("Changed "+rowsUpdated);

          if (1 == rowsUpdated)
             System.out.println(" row.");
          else
             System.out.println(" rows.");

          stmt.close();
          con.close();
       } catch( Exception e ) {
          System.out.println(e);
       }
    }
 }
```

JDBC 응용프로그램 분배

다른 Java 응용프로그램의 경우와 마찬가지로 JDBC 응용프로그램을 분배하십시오. 응용프로그램에서 DB2 클라이언트를 사용하여 DB2 서버와 통신하므로 특별히 보안에 신경쓸 필요가 없습니다. 권한 검증은 DB2 클라이언트에서 수행합니다.

클라이언트 머신에서 응용프로그램을 수행하려면 그 머신에 다음을 설치해야 합니다.

- Java 코드를 수행하는 데 필요한 Java Virtual Machine (JVM)
- A DB2 클라이언트, 여기에는 DB2 JDBC 드라이버도 포함됩니다.

응용프로그램을 빌드하려면 운영 체제용 JDK도 설치해야 합니다. Java 환경 설정, DB2 Java 응용프로그램 빌드 및 DB2 Java 응용프로그램 수행에 대해서는 *응용프로그램 빌드 안내서*에서 자세한 내용을 참조하십시오.

JDBC 애플릿 분배 및 수행

다른 Java 애플릿과 마찬가지로 인트라넷이나 인터넷과 같은 네트워크에서 JDBC 애플릿을 분배합니다. 보통 HTML 페이지에 애플릿을 Embed합니다. 예를 들어, `sqllib/samples/java`에 있는 샘플 애플릿 `DB2Applt.java`를 호출하기 위해 다음 `<APPLET>` 태그를 사용할 수 있습니다.

```
<applet code="DB2Applt.class" width=325 height=275 archive="db2java.zip">
   <param name="server" value="webhost">
   <param name="port" value="6789">
</applet>
```

애플릿을 수행하려면 단지 클라이언트 머신에 Java 지원 웹 브라우저만 있으면 됩니다. HTML 페이지를 로드할 때 애플릿 태그에서 브라우저에 애플릿 및 db2java.zip 클래스 라이브러리를 다운로드하도록 지시합니다. 이 클래스 라이브러리에는 `COM.ibm.db2.jdbc.net` 클래스에서 구현하는 DB2 JDBC 드라이버가 있습니다. 애플릿에서 JDBC API를 호출하여 DB2에 연결하면 JDBC 드라이버에서 웹 서버에서 수행 중인 JDBC 애플릿 서버를 통해 DB2 데이터베이스와 별도의 통신을 설정합니다.

주: 웹 브라우저가 서버에서 db2java.zip을 다운로드하려면 클라이언트에서 CLASSPATH 환경 변수는 db2java.zip을 포함하지 *말아야 합니다*. 애플릿은 클라이언트가 db2java.zip의 지역 버전을 사용하는 경우 제대로 기능하지 못합니다.

Java 애플릿 빌드 및 분배에 대해서는 *응용프로그램 빌드 안내서*에서 자세한 내용을 참조하십시오.

JDBC 2.0

JDBC 2.0은 최신 버전의 Sun JDBC입니다. 이 버전의 JDBC에는 두 개의 정의된 파트, **코어 API**와 **선택적 패키지 API**가 있습니다. JDBC 스펙에 대해서는 `http://www.ibm.com/software/data/db2/java/`의 DB2 Universal Database Java 웹 사이트를 참조하십시오.

운영 체제에 대해 JDBC 2.0 드라이버를 설치하는 것에 대한 자세한 내용은 *응용프로그램 빌드 안내서*을 참조하십시오.

JDBC 2.0 코어 API 지원

DB2 JDBC 2.0 드라이버에서는 JDBC 2.0 코어 API를 지원하지만 스펙에 정의된 모든 피처를 지원하는 것은 아닙니다. DB2 JDBC 2.0 드라이버는 JDBC 2.0 코어 API의 다음 기능을 지원합니다.

- Scrollable ResultSet
- `java.sql.Statement`, `java.sql.PreparedStatement` 그리고 `java.sql.CallableStatement`의 일괄처리 갱신
- `java.sql.Blob` 지원
- `java.sql.Clob` 지원

주: DB2는 저장 프로시듀어, UDF 또는 메소드에서 `java.sql.Blob`이나 `java.sql.Clob`의 사용을 지원하지 않습니다.

DB2 JDBC 2.0 드라이버는 다음 기능을 지원하지 않습니다.

- Updatable Scrollable ResultSet
- 새로운 SQL 유형(Array, Ref, Distinct, Java Object)
- 사용자 정의된 SQL 유형 맵핑

JDBC 2.0 선택적 패키지 API 지원

DB2 JDBC 2.0 드라이버는 JDBC 2.0 선택적 패키지 API의 다음 기능을 지원합니다.

명명 데이터베이스에 대한 Java 명명 및 디렉토리 인터페이스(JNDI): DB2는 Java 명명 및 디렉토리 인터페이스(JNDI)에 대해 다음과 같은 지원을 제공합니다.

javax.naming.Context

이 인터페이스는 DataSource 오브젝트의 저장과 검색을 처리하는 `COM.ibm.db2.jndi.DB2Context`으로 구현됩니다. 논리적 데이터 소스 이름의 지속 연관을 데이터베이스 이름과 같은 실제 데이터베이스 정보에서 지원하려면 이런 연관이 .db2.jndi 파일에 저장되어야 합니다. 응용프로그램에 대해 USER.HOME 환경 변수에 의해 지정된 디렉토리에 파일이 상주((또는 없으면 작성되는)합니다. 애플릿에 대해 이 파일을 웹 서버의 루트 디렉토리에 작성하여 `lookup()` 작업을 쉽게 만들어야 합니다. 애플릿은 이 클래스의 `bind()`, `rebind()`, `unbind()` 그리고 `rename()` 메소드를 지원하지 않습니다. 응용프로그램만 DataSource 오브젝트를 JNDI에 바인드할 수 있습니다.

javax.sql.Datasource

이 인터페이스는 `COM.ibm.db2.jdbc.DB2DataSource`로 구현됩니다. 이 클래스의 오브젝트를 `javax.naming.Context` 구현에 저장할 수 있습니다. 이 클래스는 또한 연결 풀링 지원을 사용합니다.

javax.naming.InitialContextFactory

이 인터페이스는 DB2Context의 인스턴스를 작성하는 `COM.ibm.db2.jndi.DB2InitialContextFactory`에서 구현됩니다. 응용프로그램은

JAVA.NAMING.FACTORY.INITIAL 환경 변수의 값을 자동으로 COM.ibm.db2.jndi.DB2InitialContextFactory로 설정합니다. 애플릿에서 이 클래스를 사용하려면 다음 구문을 사용하여 InitialContext()를 호출하십시오.

```
Hashtable env = new Hashtable( 5 );
env.put( "java.naming.factory.initial",
         "COM.ibm.db2.jndi.DB2InitialContextFactory" );
Context ctx = new InitialContext( env );
```

연결 풀링: DB2ConnectionPoolDataSource와 DB2PooledConnection은 다음과 같이 자체 연결 풀링 메소드를 구현하는 데 필요한 후크를 제공합니다.

javax.sql.ConnectionPoolDataSource

이 인터페이스는 COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource에서 구현되고 COM.ibm.db2.jdbc.DB2PooledConnection 오브젝트의 팩토리입니다.

javax.sql.PooledConnection

이 인터페이스는 COM.ibm.db2.jdbc.DB2PooledConnection으로 구현됩니다.

Java Transaction API (JTA): DB2는 DB2 JDBC 응용프로그램 드라이버를 통해 Java 트랜잭션 API(JTA)를 지원합니다. DB2는 DB2 JDBC 네트 드라이버로 JTA 지원을 제공하지 않습니다.

javax.sql.XAConnection

이 인터페이스는 COM.ibm.db2.jdbc.DB2XAConnection으로 구현됩니다.

javax.sql.XADataSource

이 인터페이스는 COM.ibm.db2.jdbc.DB2XADataSource로 구현되고 COM.ibm.db2.jdbc.DB2PooledConnection 오브젝트의 팩토리입니다.

javax.transactions.xa.XAResource

이 인터페이스는 COM.ibm.db2.jdbc.app.DBXAResource로 구현됩니다.

javax.transactions.xa.Xid

이 인터페이스는 COM.ibm.db2.jdbc.DB2Xid로 구현됩니다.

JDBC 2.0 호환성

이 버전의 스펙은 이전 버전인 1.22와 역호환성을 유지합니다. 그러나 DB2 JDBC 1.22 드라이버에서는 LOB 유형을 JDBC 1.22 스펙의 확장으로 지원하며 이 확장은 새로운 스펙의 역호환성의 일부가 되지 않습니다. 즉, JDBC 1.22 드라이버의 LOB 지원에 의존하는 기존 JDBC 응용프로그램이 새로운 드라이버에 대해 작동하지 않을 수 있습니다. LOB 및 그래픽 유형에 대한 DB2 JDBC 1.22 드라이버 지원은 750 페이지의『JDBC 1.22에 LOB 및 그래픽 오브젝트 사용』에서 자세한 내용을 참조하십시오. 문제를 해결하려면 응용프로그램을 수정하여 JDBC 2.0 드라이버에서 제공되는 LOB 지원을 사용하는 것에 대해 고려하십시오.

주: 저장 프로시듀어 또는 UDF에 있는 LOB와 그래픽 유형의 DB2 JDBC 2.0 드라이버 지원을 사용할 수 없습니다. 저장 프로시듀어나 UDF에서 LOB 또는 그래픽 유형을 사용하려면 JDBC 1.22 드라이버 지원을 사용해야 합니다.

그러나 이 솔루션이 모든 상황에 적합하지 않을 수 있습니다. 일시적인 해결책으로 JDBCVERSION 키워드를 ″122″로 설정하여 1.22 버전의 LOB 지원을 사용하도록 JDBC 2.0 드라이버에 지시할 수 있습니다. 기본값 ″200″으로 설정하면 JDBC 2.0 드라이버에서 2.0 버전의 LOB 지원을 사용합니다. db2cli.ini에서 이 키워드를 설정하거나 `getConnection` 특성 인수의 연결 속성이 있습니다.

주: JDBC 1.22 드라이버를 사용할 경우 JDBCVERSION 키워드는 JDBC에 대한 LOB 지원에 영향을 주지 않습니다.

SQLJ 프로그래밍

DB2 SQLJ 지원은 SQLJ ANSI 표준에 기초합니다. ANSI 웹 사이트와 다른 SQLJ 자원의 포인터에 대해서는 `http://www.ibm.com/software/data/db2/java`의 DB2 Java 웹 사이트를 참조하십시오. 이 장에는 SQLJ 프로그래밍에 대한 개요와 DB2 SQLJ 지원에 고유한 정보가 있습니다.

다음과 같은 SQL 구성이 SQLJ 프로그램에 나타날 수 있습니다.

- SELECT문 및 표현식과 같은 조회
- INSERT, UPDATE, DELETE와 같은 SQL 데이터 변경 명령문(DML)
- FETCH, SELECT..INTO와 같은 데이터 명령문

- COMMIT, ROLLBACK 등과 같은 트랜잭션 제어
- CREATE, DROP, ALTER와 같은 DDL, SML(Schema Manipulation Language)이라고도 함
- CALL MYPROC(:x, :y, :z)와 같은 저장 프로시듀어 호출
- VALUES(MYFUN(:x))와 같은 함수 호출

DB2 SQLJ 지원

DDB2 Application Development Client에서 제공하는 DB2 SQLJ 지원입니다. DB2 클라이언트에서 제공하는 JDBC 지원과 함께 DB2 SQLJ 지원을 통해 Java 응용프로그램, 애플릿, 저장 프로시듀어및 사용자 정의 함수(UDF)에 대해 Embedded SQL을 작성하고, 빌드하고실행할 수 있습니다. DB2 SQLJ 지원은 정적 SQL을 포함하며 DB2 데이터베이스에 바인드된 Embedded SQL문을 사용합니다.

DB2 Application Development Client에서는 다음과 같은 SQLJ 지원을 제공합니다.

- SQLJ 변환 프로그램 SQLJ는 SQLJ 프로그램의 Embedded SQL문을 Java 소스 명령문으로 대체하고 SQLJ 프로그램의 SQL 조작에 대한 정보를 포함한 직렬화된 프로파일을 생성합니다. SQLJ 변환 프로그램에서는 sqllib/java/sqlj.zip 파일을 사용합니다.
- sqllib/java/runtime.zip에서 사용 가능한 SQLJ 런타임 클래스.
- DB2 SQLJ 프로파일 사용자 정의 프로그램 db2profc는 생성된 프로파일에 저장된 SQL문을 사전 처리 컴파일하고, DB2 데이터베이스에 패키지를 생성합니다.
- DB2 SQLJ 프로파일 프린터 db2profp는 DB2 사용자 정의 프로파일 내용을 일반 텍스트로 인쇄합니다.
- SQLJ 프로파일 감사 설치 프로그램 profdb는 클래스 감사 프로그램 디버깅을 2진 프로파일의 기존 세트에 설치하거나 설치해제합니다. 일단 설치하면 응용프로그램 런타임 중 이루어지는 RTStatement 및 RTResultSet에 대한 모든 호출이 파일 또는 표준 출력에 기록됩니다. 그런 다음 파일을 검사하여 예상 수행과 추적 오류를 확인할 수 있습니다. 런타임에 기본 RTStatement 및 RTResultSet 호출 인터페이스에 대해 이루어지는 호출만이 감사됩니다.

- SQLJ 프로파일 변환 도구 profconv는 직렬화된 프로파일 인스턴스를 클래스 바이트코드 형식으로 변환합니다. 일부 브라우저에서는 애플릿과 연관된 자원 파일에서의 직렬화된 오브젝트 로드를 지원하지 않습니다. 일시적인 해결책으로 이 유틸리티를 수행하여 변환하십시오.

db2profc 및 db2profp 명령에 대해서는 *Command Reference*에서 자세한 내용을 참조하십시오. SQLJ 런타임 클래스에 대해서는 http://www.ibm.com/software/data/db2/java의 DB2 Java 웹 사이트를 참조하십시오.

DB2 SQLJ 제한사항

SQLJ로 DB2 응용프로그램을 작성할 때 다음 제한사항을 인식해야 합니다.

- DB2 SQLJ 지원은 SQL문 발행시 표준 DB2 Universal Database 제한사항을 따릅니다.
- 위치 지정된 UPDATE 및 DELETE문은 복합 SQL문의 유효한 하위 명령문이 아닙니다.
- 사전 처리 컴파일 옵션 ″DATETIME″은 지원하지 않습니다. 국제 표준 기관의 날짜 및 시간 형식만을 지원합니다.
- 사전 처리 컴파일 옵션 ″PACKAGE USING package-name″은 변환 프로그램에서 생성할 패키지의 이름을 지정합니다. 이름을 입력하지 않으면 확장자가 없고 대문자로 둘러싸인 프로파일 이름이 사용됩니다. 최대 길이는 8자입니다. SQLJ 프로파일 이름의 접미부가 _SJProfile*N*이므로 프로파일 이름은 항상 8자보다 깁니다. 여기서 *N*은 프로파일 키 번호입니다. 기본 패키지 이름은 프로파일 번호와 프로파일 키 번호의 처음 문자 8 - *pfKeyNumLen*으로 구성됩니다. 여기서 *pfKeyNumLen*은 프로파일 이름에서 프로파일 키 번호의 길입니다. 프로파일 키 번호의 길이가 7보다 길 경우 경고 없이 마지막 7자리가 사용됩니다. 예를 들어, 다음과 같습니다.

```
프로파일 이름                      기본 패키지 이름
--------------------              --------------------
App_SJProfile1                    App_SJP1
App_SJProfile123                  App_S123
App_SJProfile1234567              A1234567
App_SJProfile12345678             A2345678
```

- java.math.BigDecimal 호스트 변수를 사용할 경우 응용프로그램 변환 중 호스트 변수의 정밀도와 스케일은 사용할 수 없습니다. 명령문의 문맥에서 10진 호스트 변수의 정밀도와 스케일이 명확하지 않을 경우 CAST를 사용하여 정밀도와 스케일을 지정할 수 있습니다.
- SQL문에서 java.math.BigInteger 유형의 Java 변수는 호스트 변수로 사용할 수 없습니다.

일부 브라우저에서는 애플릿과 연관된 자원 파일에서의 직렬화된 오브젝트 로드를 지원하지 않습니다. 이런 브라우저에서 Applt 애플릿을 로드하려는 경우 다음과 같은 오류 메시지가 나타납니다.

```
java.lang.ClassNotFoundException: Applt_SJProfile0
```

일시적인 해결책으로 직렬화된 프로파일을 Java 클래스 형식으로 저장된 프로파일로 변환하는 유틸리티가 있습니다. 이 유틸리티는 sqlj.runtime.profile.util.SerProfileToClass라는 Java 클래스입니다. 이 유틸리티는 직렬화된 프로파일 자원을 입력으로 취하고 프로파일을 포함한 Java 클래스를 출력으로 생성합니다. 다음 명령을 사용하여 프로파일을 변환할 수 있습니다.

```
profconv Applt_SJProfile0.ser
```

또는

```
java sqlj.runtime.profile.util.SerProfileToClass Applt_SJProfile0.ser
```

Applt_SJProfile0.class 클래스가 그 결과로 작성됩니다. 애플릿에서 사용하는 .ser 형식의 모든 프로파일을 .class 형식의 프로파일로 대체하십시오.

SQLJ 애플릿의 경우 db2java.zip와 runtime.zip 파일이 필요합니다. db2java.zip과 runtime.zip의 클래스인 모든 애플릿 클래스를 단일 Jar 파일로 패키지화하지 않을 경우 db2java.zip과 runtime.zip(쉼표로 분리)을 "applet" 태그의 아카이브 매개변수에 넣으십시오. 아카이브 태그에서 복수 zip 파일을 지원하지 않는 브라우저의 경우 아카이브 태그에 db2java.zip을 지정한 다음 웹 브라우저에 액세스 가능한 작업 디렉토리의 애플릿 클래스로 runtime.zip을 압축해제하십시오.

Java에 SQL문 포함

SQLJ의 정적 SQL문은 *SQLJ절*에 나타납니다. SQLJ절은 Java 프로그램의 SQL 문에서 데이터베이스와 통신하는 데 사용하는 메카니즘입니다.

SQLJ 변환 프로그램에서는 다음과 같은 구조 때문에 SQLJ절과 SQL문을 인식 합니다.

- SQLJ절은 #sql 토큰으로 시작합니다.
- SQLJ절은 세미콜론으로 종료됩니다.

가장 간단한 SQLJ절은 *실행 가능 절*이며 #sql 토큰과 괄호 안의 SQL문 순으로 구성됩니다. 예를 들어, Java 명령문이 일반적으로 나타나는 위치에 다음 SQLJ절이 나타날 수 있습니다. 명령문의 목적은 TAB 테이블의 모든 행을 삭제하는 것입니다.

```
#sql { DELETE FROM TAB };
```

SQLJ 실행 가능 절에서 괄호 안에 나타나는 토큰은 호스트 변수를 제외한 SQL 토큰입니다. 변환 프로그램에서 식별할 수 있도록 모든 호스트 변수는 콜론 문자로 구분됩니다. SQL 토큰은 SQLJ 실행 가능 절의 괄호 밖에서 나타나지는 않습니다. 예를 들어, 다음 Java 메소드는 SQL 테이블에 그 인수를 삽입합니다. 메소드 본문은 호스트 변수 *x*, *y* 및 *z*를 포함하는 SQLJ 실행 가능 절로 구성됩니다.

```
void m (int x, String y, float z) throws SQLException
{
    #sql { INSERT INTO TAB1 VALUES (:x, :y, :z) };
}
```

일반적으로 SQL 토큰은 대소문자를 구분하지 않으며(이중 인용부호로 분리되는 식별자 제외) 대문자, 소문자 또는 대소문자 혼합으로 작성할 수 있습니다. 그러나 Java 토큰은 대소문자를 구분합니다. 예에서는 명확성을 위해 대소문자를 구분하지 않는 SQL은 대문자로, Java 토큰으로 소문자나 대소문자 혼합으로 표시했습니다. 이 장 전체에서 소문자 null은 Java ″널(null)″ 값을 표시하는 데 사용하며 대문자 NULL은 SQL 널(null) 값을 표시하는 데 사용합니다.

SQLJ에서의 반복기 동작 선언

테이블에서 데이터를 검색하는 SQL문과 달리 위치 지정된 UPDATE 및 DELETE 작업을 수행하는 응용프로그램 또는 보유 가능성 또는 리턴 가능성 속성의 반

복기를 사용하는 것은 두 Java 소스 파일이 필요합니다. 반복기를 하나의 소스 파일에서 공용으로 선언하고, with와 implements절을 적절히 추가하십시오.

보유 가능성 또는 리턴 가능성 속성의 값을 설정하려면, 해당 속성에 대해 with 절을 사용하여 반복기를 선언해야 합니다. 다음 예는 보유 가능성 속성을 WithHoldCurs 반복기에 대해 참으로 설정합니다.

```
#sql public iterator WithHoldCurs with (holdability=true) (String EmpNo);
```

위치 지정된 갱신을 수행하는 반복기는 sqlj.runtime.ForUpdate 인터페이스를 구현하는 implements절이 필요합니다. 예를 들어, file1.sqlj에서 다음과 같이 DelByName 반복기를 선언한다고 가정하십시오.

```
#sql public iterator DelByName implements sqlj.runtime.ForUpdate(String EmpNo);
```

그런 다음에는 다른 소스 파일에서 변환되었으며 컴파일된 반복기를 사용할 수 있습니다. 반복기를 사용하려면 다음을 실행하십시오.

1. 생성된 반복기 클래스의 인스턴스를 선언하십시오.
2. 위치 지정된 UPDATE 또는 DELETE의 SELECT문을 반복기 인스턴스에 할당하십시오.
3. 반복기를 사용하여 위치 지정된 UPDATE 또는 DELETE문을 실행하십시오.

file2.sqlj에서 위치 지정된 DELETE에 대해 DelByName을 사용하려면, 『위치 지정된 반복기를 사용하여 행 삭제』에서와 같은 명령문을 실행하십시오.

```
  {
    DelByName deliter; // Declare object of DelByName class
    String enum;
1   #sql deliter = { SELECT EMPNO FROM EMP WHERE WORKDEPT='D11'};
    while (deliter.next())
    {
2     enum = deliter.EmpNo(); // Get value from result table
3     #sql { DELETE WHERE CURRENT OF :deliter };
      // Delete row where cursor is positioned
    }
  }
```

주:

1. **1** 이 SQLJ절은 SELECT문을 실행하고, SELECT문에 대한 결과 테이블이 있는 반복기 오브젝트를 구성하고 *deliter* 변수에 반복기 오브젝트를 할당합니다.
2. **2** 이 명령문은 삭제할 행 다음에 반복기를 지정합니다.
3. **3** 이 SQLJ절은 위치 지정된 DELETE을 수행합니다.

SQLJ 예: App.sqlj

다음 예제 SQLJ 응용프로그램 App.sqlj에서는 정적 SQL을 사용하여 DB2 샘플 데이터베이스의 EMPLOYEE 테이블에서 데이터를 검색하고 갱신합니다.

1. **반복기 선언.** 이 절에서는 두 가지 유형의 반복기를 선언합니다.

   App_Cursor1

   컬럼 데이터 유형과 이름을 선언하고, 컬럼 이름에 따라 컬럼의 값을 리턴합니다(컬럼에 대한 명명된 바인딩).

   App_Cursor2

   컬럼 데이터 유형을 선언하고 컬럼 위치에 따라 컬럼의 값을 리턴합니다(컬럼에 대한 지정된 바인딩).

2. **반복기 초기화.** 반복기 오브젝트 cursor1은 조회의 결과를 사용하여 초기화됩니다. 조회는 cursor1에 결과를 저장합니다.
3. **다음 행으로 반복기 진행.** 검색할 행이 더 없을 경우 cursor1.next() 메소드는 false 부울값을 리턴합니다.
4. **데이터 이동.** 명명된 액세서 메소드 empno()는 현재 행에서 empno라는 컬럼의 값을 리턴합니다. 명명된 액세서 메소드 firstnme()은 현재 행에서 firstnme라는 컬럼의 값을 리턴합니다.
5. **호스트 변수로의 SELECT 데이터.** SELECT문은 *count1* 호스트 변수로 테이블의 행 수를 전달합니다.
6. **반복기 초기화.** 반복기 오브젝트 *cursor2*은 조회의 결과를 사용하여 초기화됩니다. 조회는 *cursor2*에 결과를 저장합니다.
7. **데이터 검색.** FETCH문은 결과 테이블의 ByPos 커서에 선언된 첫번째 컬럼의 현재 값을 *str2* 호스트 변수로 리턴합니다.

8. **FETCH..INTO문의 성공 점검.** 반복기가 행에 지정되어 있지 않은 경우 즉, 행을 가져오려는 마지막 시도가 실패한 경우 `endFetch()` 메소드는 `true` 부울값을 리턴합니다. 행을 가져오려는 마지막 시도가 성공한 경우 `endFetch()` 메소드는 `false`를 리턴합니다. `next()` 메소드를 호출하면 DB2에서 행을 가져오려 합니다. FETCH...INTO문은 `next()` 메소드를 내재적으로 호출합니다.
9. **반복기 닫기.** `close()` 메소드는 반복기에서 유지하는 자원을 해제합니다. 반복기를 명시적으로 닫아 시스템 자원이 적시에 릴리스되도록 해야 합니다.

JDBC 예: App.sqlj:

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator App_Cursor1 (String empno, String firstnme) ; 1
#sql iterator App_Cursor2 (String) ;

class App
{

  /**********************
   **  Register Driver **
   **********************/

 static
 {
   try
   {
     Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
   }
   catch (Exception e)
   {
     e.printStackTrace();
   }
 }

  /********************
   **     Main     **
   ********************/

 public static void main(String argv[])
 {
   try
   {
     App_Cursor1 cursor1;
     App_Cursor2 cursor2;

     String str1 = null;
     String str2 = null;
     long   count1;

     // URL is jdbc:db2:dbname
     String url = "jdbc:db2:sample";

     DefaultContext ctx = DefaultContext.getDefaultContext();
     if (ctx == null)
     {
       try
       {
         // connect with default id/password
         Connection con = DriverManager.getConnection(url);
         con.setAutoCommit(false);
         ctx = new DefaultContext(con);
       }
       catch (SQLException e)
       {
         System.out.println("Error: could not get a default context");
         System.err.println(e) ;
```

```
      System.exit(1);
    }
    DefaultContext.setDefaultContext(ctx);
  }

  // retrieve data from the database
  System.out.println("Retrieve some data from the database.");
  #sql cursor1 = {SELECT empno, firstnme FROM employee}; 2

  // display the result set
  // cursor1.next() returns false when there are no more rows
  System.out.println("Received results:");
  while (cursor1.next()) 3
  {
    str1 = cursor1.empno(); 4
    str2 = cursor1.firstnme();

    System.out.print (" empno= " + str1);
    System.out.print (" firstname= " + str2);
    System.out.print ("");
  }
  cursor1.close(); 9

  // retrieve number of employee from the database
  #sql { SELECT count(*) into :count1 FROM employee }; 5
  if (1 == count1)
    System.out.println ("There is 1 row in employee table");
  else
    System.out.println ("There are " + count1
                        + " rows in employee table");

  // update the database
  System.out.println("Update the database. ");
  #sql { UPDATE employee SET firstnme = 'SHILI' WHERE empno = '000010' };

  // retrieve the updated data from the database
  System.out.println("Retrieve the updated data from the database.");
  str1 = "000010";
  #sql cursor2 = {SELECT firstnme FROM employee WHERE empno = :str1}; 6

  // display the result set
  // cursor2.next() returns false when there are no more rows
  System.out.println("Received results:");
  while (true)
  {
    #sql { FETCH :cursor2 INTO :str2 }; 7
    if (cursor2.endFetch()) break; 8

    System.out.print (" empno= " + str1);
    System.out.print (" firstname= " + str2);
    System.out.print ("");
  }
  cursor2.close(); 9

  // rollback the update
  System.out.println("Rollback the update.");
  #sql { ROLLBACK work };
  System.out.println("Rollback done.");
}
catch( Exception e )
```

```
      {
        e.printStackTrace();
      }
    }
  }
```

Java 호스트 변수

Embedded SQL문의 인수는 *호스트 변수*, SQL문에 나타난 호스트 언어의 변수를 통해 전달됩니다. 호스트 변수에는 최대 세 가지 부분이 있습니다.

- 콜론(:) 접두부
- IN, OUT 또는 INOUT과 같은 선택적 매개변수 모드 식별자
- 매개변수, 변수 또는 필드에 대한 Java 식별자인 Java 호스트 변수

Java 식별자의 평가는 Java 프로그램에 부가 작용을 미치지 않으므로, SQLJ 클래스를 대체하기 위해 생성된 Java 코드에 여러번 나타날 수 있습니다.

다음 조회에는 *:x* 호스트 변수가 있으며 이 호스트 변수는 조회를 포함한 범위에 가시적인 Java 변수, 필드 또는 매개변수 *x*입니다.

```
SELECT  COL1,  COL2  FROM  TABLE1  WHERE  :x  > COL3
```

복합 SQL에 지정된 모든 호스트 변수는 기본적으로 입력 호스트 변수입니다. 호스트 변수를 출력 호스트 변수로 표시하려면 호스트 변수 앞에 매개변수 모드 식별자 OUT 또는 INOUT을 지정해야 합니다. 예를 들어, 다음과 같습니다.

```
#sql {begin compound atomic static
     select count(*) into :OUT count1 from employee;
     end compound}
```

SQLJ에 있는 저장 프로시듀어 및 함수에 대한 호출

데이터베이스에는 *저장 프로시듀어*, *사용자 정의 함수* 그리고 *사용자 정의 메소드*가 들어 있습니다. 저장 프로시듀어, 사용자 정의 함수 그리고 사용자 정의 메소드는 데이터베이스에서 실행되는 명명된 스키마 오브젝트입니다. Java 명령문으로 나타나는 SQLJ 실행 가능 절에서 다음과 같은 CALL문을 통해 저장 프로시듀어를 호출할 수 있습니다.

```
#sql { CALL SOME_PROC(:INOUT myarg) };
```

저장 프로시듀어에는 IN, OUT 또는 INOUT 매개변수가 들어갈 수 있습니다. 위의 경우에 *myarg* 호스트 변수의 값은 그 절의 실행으로 변경됩니다. SQLJ 실행 가능 절에서 SQL VALUES 구성을 통해 함수를 호출할 수 있습니다. 예를 들어, 정수를 리턴하는 함수 F를 가정하십시오. 다음 예는 그 함수의 호출을 나타낸 다음 Java 지역 변수 *x*에 그 결과를 할당합니다.

```
{
    int x;
    #sql x = { VALUES( F(34) ) };
}
```

SQLJ 프로그램 컴파일 및 수행

MyClass 프로그램 이름으로 SQLJ 프로그램을 수행하려면 다음을 실행하십시오.

1. Embedded SQL로 Java 소스 코드를 변환하여 MyClass.java Java 소스 코드와 MyClass_SJProfile0.ser, MyClass_SJProfile1.ser, ... 프로파일을 생성하십시오(각 연결 문맥에 대해 한 프로파일씩).

   ```
   sqlj MyClass.sqlj
   ```

   sqlj.properties 파일을 지정하지 않고 SQLJ 변환 프로그램을 사용하면 변환 프로그램에서 다음 값을 사용합니다.

   ```
   sqlj.url=jdbc:db2:sample
   sqlj.driver=COM.ibm.db2.jdbc.app.DB2Driver
   sqlj.online=sqlj.semantics.JdbcChecker
   sqlj.offline=sqlj.semantics.OfflineChecker
   ```

   sqlj.properties 파일을 지정할 경우 다음 옵션이 설정되어 있는지 확인하십시오.

   ```
   sqlj.url=jdbc:db2:dbname
   sqlj.driver=COM.ibm.db2.jdbc.app.DB2Driver
   sqlj.online=sqlj.semantics.JdbcChecker
   sqlj.offline=sqlj.semantics.OfflineChecker
   ```

   여기서 *dbname*은 데이터베이스의 이름입니다. 또한 명령행에서 이런 옵션을 지정할 수도 있습니다. 예를 들어, MyClass를 변환할 때 mydata 데이터베이스를 지정하기 위해 다음 명령을 발행할 수도 있습니다.

   ```
   sqlj -url=jdbc:db2:mydata MyClass.sqlj
   ```

`-compile=false`절로 컴파일 옵션을 명시적으로 설정해제하지 않는 한 SQLJ 변환 프로그램에서 클래스 파일로 소스 코드를 자동으로 변환한다는 점에 주의하십시오.

2. 생성된 프로파일에서 DB2 SQLJ 사용자 정의 프로그램을 설치한 다음 DB2 데이터베이스 *dbname*에서 DB2 패키지를 작성하십시오.

```
db2profc -user=user-name -password=user-password -url=jdbc:db2:dbname
  -prepoptions="bindfile using MyClass0.bnd package using MyClass0"
         MyClass_SJProfile0.ser
db2profc -user=user-name -password=user-password -url=jdbc:db2:dbname
  -prepoptions="bindfile using MyClass1.bnd package using MyClass1"
         MyClass_SJProfile1.ser
...
```

3. SQLJ 프로그램을 실행하십시오.

```
java MyClass
```

변환 프로그램에서는 SQLJ이 사용자 정의된 데이터베이스에 대한 SQL 구문을 생성합니다. 예를 들면 다음과 같습니다.

```
i = { VALUES (  F(:x) ) };
```

이 구문은 SQLJ 변환 프로그램으로 변환되며

```
? = VALUES (F (?))
```

생성된 프로파일에 저장됩니다. DB2 Universal Database 데이터베이스에 연결하면 DB2에서 VALUE문을 다음으로 사용자 정의합니다.

```
VALUES(F(?)) INTO ?
```

그러나 OS/390용 DB2 Universal Database 데이터베이스에 연결하면 DB2에서 VALUE문을 다음으로 사용자 정의합니다.

```
SELECT F(?) INTO ? FROM SYSIBM.SYSDUMMY1
```

DB2 Universal Database 데이터베이스에 대해 DB2 SQLJ 프로파일 사용자 정의 프로그램 `db2profc`를 수행하여 바인드 파일을 생성할 경우 바인드 파일에 `VALUES`절이 있으면 바인드 파일을 사용하여 OS/390용 DB2 데이터베이스에 바인드할 수 없습니다. 이런 내용은 OS/390용 DB2 데이터베이스에 대해 바인드 파일을 생성하여 DB2 Universal Database 데이터베이스에 바인드하려는 경우에도 적용됩니다.

DB2 SQLJ 프로그램 빌드 및 수행에 대해서는 *응용프로그램 빌드 안내서*에서 자세한 내용을 참조하십시오.

SQLJ 변환기 옵션

SQLJ 변환기는 DB2 PRECOMPILE 명령과 같은 사전 처리 컴파일 옵션을 지원합니다. 다음은 예외사항입니다.

```
CONNECT
DISCONNECT
DYNAMICRULES
NOLINEMACRO
OPTLEVEL
OUTPUT
SQLCA
SQLFLAG
SQLRULES
SYNCPOINT
TARGET
WCHARTYPE
```

SQLJ 변환 프로그램에서 생성한 프로파일의 내용을 일반 텍스트로 인쇄하려면 다음과 같이 profp 유틸리티를 사용하십시오.

```
profp MyClass_SJProfile0.ser
profp MyClass_SJProfile1.ser
  ...
```

프로파일의 DB2 사용자 정의 버전의 내용을 일반 텍스트로 인쇄하려면 db2profp 유틸리티를 다음과 같이 사용하십시오. 여기서 *dbname*은 데이터베이스의 이름입니다.

```
db2profp -user=user-name -password=user-password -url=jdbc:db2:dbname
         MyClass_SJProfile0.ser
db2profp -user=user-name -password=user-password -url=jdbc:db2:dbname
         MyClass_SJProfile1.ser
    ...
```

Java에 있는 저장 프로시듀어 및 UDF

다른 프로그래밍 언어의 경우와 마찬가지로 저장 프로시듀어와 UDF를 Java로 작성하고 사용할 수 있습니다. Java 코드를 작성할 때 알아야 하는 몇 가지 프로그래밍 고려사항(749 페이지의 『Java 함수 정의』 참조)을 참조하십시오. 또한 저장 프로시듀어와 UDF를 등록해야 합니다. 저장 프로시듀어 등록 방법에 대해서는 207 페이지의 『제7장 저장 프로시듀어』에서 자세한 내용을 참조하십시오. UDF 등록 방법에 대해서는 *SQL 참조서*의 CREATE FUNCTION문을 참조하십시오.

서버에서 UDF와 저장 프로시듀어를 수행하려면 DB2에서 JVM을 호출합니다. 데이터베이스를 시작하기 전에 적합한 JDK(Java Development Kit)이나 Java Runtime Environment가 DB2 서버에 설치되고 구성되어 있는지 확인하십시오.

JVM 런타임 라이브러리는 시스템 검색 경로인 PATH 또는 LIBPATH 또는 LD_LIBRARY_PATH 및 CLASSPATH에서 사용 가능해야 합니다. Java 환경 설정에 대해서는 *응용프로그램 빌드 안내서*에서 자세한 내용을 참조하십시오.

처음으로 Java UDF나 저장 프로시듀어를 호출할 때 DB2에서 JVM을 로드하거나 시작합니다. NOT FENCED UDF와 저장 프로시듀어에 대해 DB2는 데이터베이스 인스턴스마다 JVM을 하나 로드하고 데이터베이스 엔진의 주소 공간에서 이를 실행하여 성능을 향상시킵니다. 분리 UDF의 경우 DB2는 `db2udf` 프로세스에서 명확한 JVM을 사용합니다. 마찬가지로 분리 저장 프로시듀어는 `db2dari` 프로세스에서 명확한 JVM을 사용합니다. 모든 경우에 Embedding 프로세스가 종료할 때까지 JVM은 여전히 로드되어 있습니다.

주: *지역 클라이언트가 있는 데이터베이스 서버* 노드 유형을 수행할 경우 Java 저장 프로시듀어를 호출하기 전에 `maxdari` 데이터베이스 관리 프로그램 구성 매개변수를 0이 아닌 값으로 설정해야 합니다.

`sqllib/samples/java` 디렉토리에 제공된 Java 저장 프로시듀어를 연구할 수 있습니다. DB2에 포함된 샘플 프로그램의 목록에 대해서는 835 페이지의 『부록B. 샘플 프로그램』에서 자세한 내용을 참조하십시오.

저장 프로시듀어 또는 UDF를 구현하는 데 사용하는 모든 Java 클래스 파일은 데이터베이스에 설치한 JAR 파일이나 『Java 클래스를 넣을 위치』에서 언급한 운영 체제의 올바른 저장 프로시듀어 또는 UDF 경로에 있어야 합니다.

주: 혼합 코드 페이지 데이터베이스 서버에서, 대형 혼합 코드 페이지 문자열의 문자 경계에 대한 랜덤한 액세스를 아직 구현하지 않았으므로 Java 사용자 정의 함수와 저장 프로시듀어에서 CLOB 유형 인수를 사용할 수 없습니다. SBCS 데이터베이스에 대해 모든 LOB 유형을 완전히 지원합니다. 혼합 데이터베이스의 경우에는 BLOB 및 DBCLOB 유형에 대해 지원합니다. 일시적인 해결책으로, 혼합 데이터베이스 시스템에서 수행 중인 응용프로그램에서 CLOB 인수를 DBCLOB, LONG VARGRAPHIC 또는 LONG VARCHAR 유형으로 변환해야 합니다. UDF의 경우 CAST 연산자로 이를 실행할 수 있습니다.

Java 클래스를 넣을 위치

저장 프로시듀어 또는 UDF에 대해 개별 Java 클래스 파일을 선택하거나 클래스 파일을 JAR 파일로 수집해 넣고 데이터베이스에 JAR 파일을 설치할 수 있습니다. JAR 파일을 사용하려는 경우 자세한 지침은 746 페이지의 『Java 저장 프로시듀어 및 UDF』의 내용을 참조하십시오.

주: Java 루틴 클래스 파일을 갱신하거나 대체하면, CALL SQLJ.REFRESH_CLASSES()문을 실행하여 DB2에서 갱신된 클래스를 로드해야 합니다. CALL SQLJ.REFRESH_CLASSES() 문에 대한 자세한 내용은 742 페이지의 『루틴에서 Java 클래스 갱신』을 참조하십시오.

DB2를 사용하여 저장 프로시듀어와 UDF 찾아 사용하려면 다음과 같이 운영 체제에 대해 정의된 디렉토리인 *함수 디렉토리*에 해당 클래스 파일을 저장해야 합니다.

Unix 운영 체제

```
sqllib/function
```

OS/2 또는 Windows 32 비트 운영 체제

instance_name\function, 여기서 *instance_name*은 *DB2INSTPROF* 인스턴스 고유의 레지스트리 설정 값을 표시합니다.

예를 들어, C:\sqllib에 DB2가 설치되어 있고 *DB2INSTPROF* 레지스트리 설정이 지정되어 있지 않은 Windows NT 서버의 함수 디렉토리는 다음과 같습니다.

```
C:\sqllib\function
```

개별 클래스 파일을 사용하려는 경우 사용자 운영 체제의 해당 디렉토리에 클래스 파일을 저장해야 합니다. 클래스를 Java 패키지의 일부로 선언한 경우 함수 디렉토리에 해당 서브디렉토리를 작성한 다음 해당 서브디렉토리에 파일을 넣으십시오. 예를 들어, Linux 시스템용 ibm.tests.test1 클래스를 작성할 경우 sqllib/function/ibm/tests에 해당 Java 바이트코드 파일 test1.class를 저장하십시오.

DB2에서 호출하는 JVM은 CLASSPATH 환경 변수를 사용하여 Java 파일을 찾습니다. DB2에서는 함수 디렉토리와 sqllib/java/db2java.zip을 CLASSPATH 설정 앞에 추가합니다.

JVM에서 Java 클래스 파일을 찾을 수 있도록 환경을 설정하려면 jdk11_path 구성 매개변수를 설정하거나 기본값을 사용해야 합니다. 또한 사용자 응용프로그램의 힙(heap)을 증가시키도록 java_heap_sz 구성 매개변수를 설정해야 할 수 있습니다. 구성 매개변수에 대해서는 *관리 안내서*의 내용을 참조하십시오.

루틴에서 Java 클래스 갱신

Java 루틴 클래스를 갱신할 때 CALL SQLJ.REFRESH_CLASSES()문을 실행하여 DB2에서 새 클래스를 로드하게 해야 합니다. Java 루틴 코드를 갱신한 후에 CALL SQLJ.REFRESH_CLASSES()문을 실행하지 않으면 DB2는 계속 클래스의 이전 버전을 사용합니다. CALL SQLJ.REFRESH_CLASSES()문은 FENCED 루틴에만 적용됩니다. DB2는 COMMIT 또는 ROLLBACK이 발생할 때 클래스를 새로 고칩니다.

주: 데이터베이스 관리자를 중지하고 다시 시작하지 않은 상태에서는 NOT FENCED 루틴을 갱신할 수 없습니다.

Java에서 저장 프로시듀어 디버깅

저장 프로시듀어가 AIX 또는 Windows NT서버에서 실행할 경우 DB2에서는 JDBC로 작성한 저장 프로시듀어를 대화식으로 디버그하는 기능을 제공합니다. DB2 Stored Procedure Builder를 사용하면 가장 쉽게 디버깅을 호출할 수 있습니다. 그 실행 방법에 대해서는 Stored Procedure Builder에 대한 온라인 도움말을 참조하십시오.

이 절에서는 다음 주제를 다룹니다.

- 디버그 준비
- 디버그 테이블 데이터 상주
- 디버거 호출

디버그 준비

1. JDK 개발 환경 문서에 따라 저장 프로시듀어를 디버그 모드로 컴파일하십시오.
2. 서버 준비.
   - 소스 코드가 서버에 저장되어 있는 경우 741 페이지의 『Java 클래스를 넣을 위치』에서 정의한 함수 디렉토리에 소스 코드를 저장하거나 Java 소스 코드 디렉토리를 포함하도록 CLASSPATH 환경 변수를 설정하십시오.
   - db2set 명령을 사용하여 인스턴스 디버깅을 사용하십시오.

     ```
     db2set DB2ROUTINE_DEBUG=ON
     ```
3. 클라이언트 환경 변수를 설정하십시오.
   - 소스 코드가 클라이언트에 저장되어 있는 경우 저장 프로시듀어의 소스 코드가 있는 디렉토리로 DB2_DBG_PATH 환경 변수를 설정하십시오.
4. 디버그 테이블 작성.

   Stored Procedure Builder를 사용하여 디버거를 호출하지 않을 경우 다음 명령으로 디버그 테이블을 작성하십시오.

   ```
   db2 -tf sqllib/misc/db2debug.ddl
   ```

   주: DB2 Enterprise - Extended Edition 시스템에서 기본 노드그룹은 USERSPACE1 테이블 공간에 대한 IBMDEFAULTGROUP이며 시스템

에 대해 정의된 모든 노드로 구성됩니다. DB2 Enterprise - Extended Edition 구성에서 저장 프로시듀어 디버깅의 성능을 향상시키려면 디버깅이 발생한 단일 조정자 노드가 있어야 하며 그 노드만 있는 노드 그룹을 정의해야 합니다.

5. 클라이언트에서 디버거 디먼을 시작하십시오.

   저장 프로시듀어 클라이언트에서 다음 명령으로 디버거 디먼을 시작하십시오.

   ```
   db2dbugd -qport=portno
   ```

   여기서 *portno*는 사용되지 않는 TCP/IP 포트 번호입니다. 값을 제공하지 않을 경우 디버거에서는 8000을 기본 포트 번호로 사용합니다. Windows 32 비트 운영 체제에서 DB2 폴더에 있는 디버거 디먼 단축 아이콘을 클릭하여 디폴트 포트 번호로 디버거를 시작할 수도 있습니다.

디버그 테이블 데이터 상주

디버그 테이블에는 사용자가 디버그한 저장 프로시듀어 및 디버그를 실행한 클라이언트/서버 환경에 대한 정보가 있습니다. INSERT, UPDATE 또는 DELETE 특권이 있는 사용자나 DBA가 기본 테이블 DB2DBG.ROUTINE_DEBUG의 값을 직접 조작할 수 있습니다. 그러나 DBA에서 제한사항을 더 추가하지 않는 한 누구나 사용자 뷰 DB2DBG.ROUTINE_DEBUG_USER를 통해 행을 추가하거나 갱신하거나 삭제할 수 있습니다. 따라서 이 절의 나머지 부분에서는 사용자 뷰를 통해 테이블에 데이터를 상주한다고 가정합니다.

Stored Procedure Builder를 사용하여 디버깅을 호출할 경우 디버거 유틸리티를 사용하여 디버그 테이블의 데이터를 상주하고 테이블을 관리할 수 있습니다. 그렇지 않은 경우 지정한 저장 프로시듀어에 대한 디버깅 지원을 사용하려면 CLP에서 다음 명령을 발행하십시오.

```
DB2 INSERT INTO db2dbg.routine_debug_user (AUTHID, TYPE,
   ROUTINE_SCHEMA, SPECIFICNAME, DEBUG_ON, CLIENT_IPADDR)
VALUES ('authid', 'S', 'schema', 'proc_name', 'Y', 'IP_num')
```

여기서, :

authid

저장 프로시듀어 디버깅에 사용되는 사용자 이름. 즉, 데이터베이스에 연결하는 데 사용되는 사용자 이름.

schema

저장 프로시듀어의 스키마 이름.

proc_name

저장 프로시듀어의 특정 이름. 특정 이름을 제공하지 않은 경우 CREATE PROCEDURE 명령 또는 시스템 생성 식별자에 제공된 특정 이름입니다.

IP_num

저장 프로시듀어를 디버그하는 데 사용한 클라이언트의 IP 주소로 그 양식은 `nnn.nnn.nnn.nnn`.

예를 들어, 123.234.111.222 IP 주소에 있는 디버깅 클라이언트를 사용하여 *USER1* 사용자로 *MySchema.myProc* 저장 프로시듀어에 대한 디버깅을 사용하려면 다음 명령을 입력하십시오.

```
DB2 INSERT INTO db2dbg.routine_debug_user (AUTHID, TYPE,
   ROUTINE_SCHEMA, SPECIFICNAME, DEBUG_ON, CLIENT_IPADDR)
VALUES ('USER1', 'S', 'MySchema', 'myProc', 'Y', '123.234.111.222')
```

저장 프로시듀어를 제거할 경우 그 디버그 정보는 디버그 테이블에서 자동으로 삭제되지 않습니다. 존재하지 않는 저장 프로시듀어에 대한 디버그 정보로는 사용자 데이터베이스나 인스턴스에 해로운 영향을 줄 수 없습니다. DB2 카탈로그로 동기화된 디버그 테이블을 유지하려면 디버그 정보를 수작업으로 삭제해야 합니다.

디버그 테이블을 수작업으로 또는 Stored Procedure Builder를 사용하여 작성하든 디버그 테이블의 이름은 DB2DBG.ROUTINE_DEBUG이며 그 정의는 다음과 같습니다.

표 32. DB2DBG.ROUTINE_DEBUG 테이블 정의

컬럼 이름	데이터 유형	속성	설명
AUTHID	VARCHAR(128)	NOT NULL, DEFAULT USER	이 저장 프로시듀어에 대한 디버깅이 수행되는 응용프로그램 authid. 데이터베이스에 연결시 제공되는 사용자 ID입니다.
TYPE	CHAR(1)	NOT NULL	유효한 값: 'S'(저장 프로시듀어)
ROUTINE_SCHEMA	VARCHAR(128)	NOT NULL	디버그할 저장 프로시듀어의 스키마 이름
SPECIFICNAME	VARCHAR(18)	NOT NULL	디버그할 저장 프로시듀어의 특정 이름

표 32. DB2DBG.ROUTINE_DEBUG 테이블 정의 (계속)

컬럼 이름	데이터 유형	속성	설명
DEBUG_ON	CHAR(1)	NOT NULL, DEFAULT 'N'	유효한 값: • **Y** - ROUTINE_SCHEMA.SPECIFICNAME에 명명된 저장 프로시듀어에 대한 디버깅 사용 • **N** - ROUTINE_SCHEMA.SPECIFICNAME에 명명된 저장 프로시듀어에 대한 디버깅 사용안함. 기본값입니다.
CLIENT_IPADDR	VARCHAR(15)	NOT NULL	nnn.nnn.nnn.nnn 양식의 디버깅을 실행하는 클라이언트의 IP 주소
CLIENT_PORT	INTEGER	NOT NULL, DEFAULT 8000	디버깅 통신의 포트. 기본값은 8000입니다.
DEBUG_STARTN	INTEGER	NOT NULL	사용되지 않음.
DEBUG_STOPN	INTEGER	NOT NULL	사용되지 않음.
이 테이블의 기본 키는 AUTHID, TYPE, ROUTINE_SCHEMA, SPECIFICNAME입니다.			

디버거 호출

위의 단계를 따른 경우 저장 프로시듀어를 호출하면 디버그 테이블에 지정한 IP 주소로 클라이언트의 디버거가 호출됩니다.

디버거에서 소스 코드를 step through하고, 변수를 표시하고, 소스 코드에 중단점을 설정할 수 있습니다. 디버거 사용에 대해서는 온라인 도움말에 포함된 디버거 문서를 참조하십시오.

Java 저장 프로시듀어 및 UDF

*Java 루틴*으로 알려진 Java 저장 프로시듀어와 UDF는 DB2 카탈로그에 등록되어야 합니다. DB2 Universal Database 버전 7에서는 Java 함수 루틴 및 전개를 위해 *SQLJ 루틴* 코어 스펙을 지원합니다. CREATE PROCEDURE 및 CREATE FUNCTION문에서 PARAMETER STYLE JAVA를 사용하여 SQLJ 루틴을 따르도록 지정하십시오.

또는 DB2는 DB2 V5 및 V5.2 PARAMETER STYLE DB2GENERAL 저장 프로시듀어와 UDF를 지원합니다. 857 페이지의 『부록C. DB2DARI 및 DB2GENERAL 저장 프로시듀어와 UDF』에서 자세한 내용을 참조하십시오.

Java 함수나 저장 프로시듀어를 등록하려면 다음 단계를 따르십시오.

1. Java 메소드를 Java 루틴으로 작성하십시오. Java 소스 코드를 Java 클래스 파일로 컴파일하십시오. Java 저장 프로시듀어 작성에 대해서는 207 페이지의 『제7장 저장 프로시듀어』에서 자세한 내용을 참조하십시오. Java UDF 작성에 대해서는 466 페이지의 『Java 사용자 정의 함수 작성 및 사용』에서 자세한 내용을 참조하십시오.
2. *jar* 파일에 Java 루틴을 포함한 클래스 파일을 수집하십시오. 단일 JAR 파일에 클래스 파일을 하나 이상 수집할 수 있습니다. JAR 파일 작성에 대한 지침은 *응용프로그램 빌드 안내서*의 내용을 참조하십시오.
3. DB2 인스턴스에 JAR 파일을 설치하십시오. 명령행에서 cALL SQLJ.INSTALL_JAR문을 사용하는 방법에 대해서는 『JAR 파일 설치, 대체 및 제거』에서 자세한 내용을 참조하십시오. 또한 CLP로부터 응용프로그램의 `sqlj.install_jar` 프로시듀어를 호출할 수도 있습니다.
4. Java 루틴의 해당 CREATE PROCEDURE 또는 CREATE FUNCTION SQL 문을 발행하십시오.
   - CREATE PROCEDURE문 사용에 대한 설명과 예는 214 페이지의 『저장 프로시듀어 레지스터』의 내용을 참조하십시오.
   - CREATE FUNCTION문 사용에 대한 설명과 예는 *SQL 참조서*의 내용을 참조하십시오.

JAR 파일을 설치하면 DB2에서 JAR 파일로부터 Java 클래스 파일을 추출한 다음 시스템 카탈로그의 각 클래스를 등록합니다. DB2에서 JAR 파일을 함수 디렉토리의 `jar/`*`schema`* 서브디렉토리로 복사합니다. DB2에서는 *jar-id*절에 지정된 이름을 JAR 파일의 새로운 사본에 지정합니다. DB2 인스턴스에 설치한 JAR 파일을 직접 수정하지 마십시오. 대신 CALL SQLJ.REMOVE_JAR and CALL SQLJ.REPLACE_JAR 명령을 사용하여 설치된 JAR 파일을 제거하거나 대체할 수 있습니다.

JAR 파일 설치, 대체 및 제거

DB2 인스턴스에서 JAR 파일을 설치하거나 대체하기 위해 명령행 처리기에서 다음 명령 구문을 사용할 수 있습니다.

```
                                  (1)                     (2)
►►─CALL─┬─SQLJ.INSTALL_JAR─┬──(─'─jar-url──────'─,─'─jar-id──────'─)──────────►◄
        └─SQLJ.REPLACE_JAR─┘
```

주:

1 설치하거나 제거할 JAR 파일이 있는 URL을 지정합니다. 지원되는 유일한 URL 스킴은 'file:'입니다.

2 jar-url로 지정한 파일과 연관시킬 데이터베이스에서 JAR 식별자를 지정합니다.

주: OS/2 및 Windows 32 비트 운영 체제에서 DB2는 *DB2INSTPROF* 인스턴스 고유의 레지스트리 설정으로 지정된 경로에 JAR 파일을 저장합니다. 인스턴스에 대해 JAR 파일을 고유하게 유지하려면 그 인스턴스의 *DB2INSTPROF* 에 대해 고유 값을 지정해야 합니다.

예를 들어, DB2 인스턴스의 `file:/home/db2inst/classes/` 디렉토리에 있는 `Proc.jar` 파일을 설치하려면 명령행 처리기에서 다음 명령을 발행하십시오.

```
CALL SQLJ.INSTALL_JAR('file:/home/db2inst/classes/Proc.jar' , 'myproc_jar')
```

`Procedure.jar` 파일을 사용하는 후속 SQL 명령에서는 `myproc_jar` 이름으로 이를 참조합니다. 데이터베이스에서 JAR 파일을 제거하려면 다음 구문으로 CALL REMOVE_JAR 명령을 사용하십시오.

```
                                  (1)
►►─CALL─SQLJ.REMOVE_JAR──(─'─jar-id──────'─)─────────────────────────────────►◄
```

주:

1 데이터베이스에서 제거할 JAR 파일의 JAR 식별자를 지정합니다.

데이터베이스에서 `myProc_jar` JAR 파일을 제거하려면 명령행 처리기에서 다음 명령을 입력하십시오.

```
CALL SQLJ.REMOVE_JAR('myProc_jar')
```

Java 함수 정의

Java 루틴을 작성하려면 public 클래스에서 해당 public static 메소드를 코딩해야 합니다. 또한 Java 루틴은 throws SQLException절로선언해야 합니다. 메소드 본문으로부터 예상되는 출력과 일치하도록 메소드 서명과 메소드 선언의 나머지를 코딩하십시오.

Java에서 값을 리턴하지 않는 함수: 호출 프로그램으로 값을 리턴하지 않는 메소드를 작성하려면 void를 리턴하도록 메소드를 선언한 다음 메소드 본문으로 전달해야 하는 서명에 임의의 매개변수를 포함하십시오. 단순 UDPATE를 수행한 다음 클라이언트 응용프로그램에 값을 리턴하지 않는 저장 프로시듀어를 다음과 같이 작성할 수 있습니다.

```
public class JavaExamples {
      public static void updateJob(String oldJob, String newJob)
            throws SQLException {
         Connection conn=DriverManager.getConnection("jdbc:ibm.db2.sample");
         PreparedStatement stmt = conn.prepareStatement("UPDATE employee
            SET job = ? WHERE job = ?");
         stmt.setString(1, newJob);
         stmt.setString(2, oldJob);
         stmt.executeUpdate();
         conn.close();
         return;
         }
      }
```

Java에서 단일 값을 리턴하는 함수: 각 SQL 데이터 유형에 해당하는 Java 리턴 유형으로 단일 값을 리턴하는 Java 메소드를 선언하십시오(712 페이지의 『Java에서 지원되는 SQL 데이터 유형』 참조). 다음과 같이 SQL INTEGER 값을 리턴하는 스칼라 UDF를 작성할 수 있습니다.

```
public class JavaExamples {
   public static int getDivision(String division) throws SQLException {
      if (division.equals("Corporate")) return 1;
      else if (division.equals("Eastern")) return 2;
      else if (division.equals("Midwest")) return 3;
      else if (division.equals("Western")) return 4;
      else return 5;
   }
}
```

Java에서 복수 값을 리턴하는 함수: 저장 프로시듀어로 카탈로그화된 Java 메소드에서 값을 하나 이상 리턴할 수 있습니다. 또한 복수 결과 세트를 리턴하는 Java 저장 프로시듀어를 작성할 수도 있습니다. 251 페이지의 『저장 프로시듀어

로부터 결과 세트 리턴』에서 자세한 내용을 참조하십시오. 사전 결정된 값의 수를 리턴하는 메소드를 코딩하려면 void 리턴 유형을 선언한 다음 메소드 서명에 예상된 출력의 유형을 배열로 포함하십시오. 이름, 서비스 연도 및 지정된 임계값 아래의 급여를 받는 두 명의 최장년 직원의 급여를 리턴하는 저장 프로시듀어를 다음과 같이 작성할 수 있습니다.

```
public Class JavaExamples {
   public static void lowSenioritySalary
      (String[] name1, int[] years1, BigDecimal[] salary1,
       String[] name2, int[] years2, BigDecimal[] salary2,
       Integer threshhold) throws SQLException {
      #sql iterator ByNames (String name, int years, BigDecimal salary);
      ByNames result;
      #sql result = {"SELECT name, years, salary
                      FROM staff
                      WHERE salary < :threshhold
                      ORDER BY years DESC"};
      if (result.next()) {
         name1[0] = result.name();
         years1[0] = result.years();
         salary1[0] = result.salary();
      }
      else {
         name1[0] = "****";
         return;
      }
      if (result.next()) {
         name2[0] = result.name();
         years2[0] = result.years();
         salary2[0] = result.salary();
      }
      else {
         name2[0] = "****";
         return;
      }
   }
}
```

JDBC 1.22에 LOB 및 그래픽 오브젝트 사용

JDK 1.2에 대한 JDBC 2.0 스펙은 LOB 및 그래픽 유형에 대한 지원을 정의합니다. DB2 JDBC 2.0 드라이버 지원에 대해서는 723 페이지의『JDBC 2.0』에서 자세한 내용을 참조하십시오.

주: 저장 프로시듀어 또는 UDF에 있는 LOB와 그래픽 유형의 DB2 JDBC 2.0 드라이버 지원을 사용할 수 없습니다. 저장 프로시듀어나 UDF에서 LOB 또는 그래픽 유형을 사용하려면, JDBC 1.22 LOB 지원을 사용해야 합니다. DB2

JDBC 2.0 드라이버로 DB2 JDBC 1.22 LOB 지원하기에 대한 자세한 내용은 726 페이지의 『JDBC 2.0 호환성』을 참조하십시오.

그러나 JDBC 1.22 스펙에서는 대형 오브젝트(LOB)나 그래픽 유형을 명시적으로 언급하지 않습니다. JDBC 1.22 드라이버를 사용할 경우 DB2에서는 LOB 및 그래픽 유형을 다음과 같이 지원합니다.

응용프로그램에서 LOB 또는 그래픽 유형을 사용할 경우 LOB를 해당 LONGVAR 유형으로 처리하십시오. SQL에서 LOB 유형이 최대 길이로 선언되므로 선언된 한계보다 긴 문자열이나 배열을 리턴하지 않아야 합니다. 이 고려사항은 SQL 문자열 유형에도 적용됩니다.

GRAPHIC과 DBCLOB 데이터 유형을 해당 CHAR 유형으로 처리하십시오.

서버 코드 페이지에서 유니코드로 데이터를 변환하려면 DB2 클라이언트는 먼저 데이터를 서버 코드 페이지에서 클라이언트 코드 페이지로 변환합니다. 그러면 클라이언트를 클라이언트 코드 페이지에서 유니코드로 데이터를 변환합니다. 다음 JDBC API는 데이터를 유니코드와 상호 변환합니다.

getString
: 서버 코드 페이지에서 유니코드로 변환합니다.

setString
: 유니코드에서 서버 코드 페이지로 변환합니다.

getUnicodeStream
: 서버 코드 페이지에서 유니코드로 변환합니다.

setUnicodeStream
: 유니코드에서 서버 코드 페이지로 변환합니다.

다음 JDBC API에는 클라이언트 코드 페이지와 서버 코드 페이지간의 변환이 일어납니다.

setAsciiStream
: 클라이언트 코드 페이지에서 서버 코드 페이지로 변환합니다.

getAsciiStream
: 서버 코드 페이지에서 클라이언트 코드 페이지로 변환합니다.

JDBC 및 SQLJ 상호운용성

SQLJ 언어는 프로그램이 작성된 시기에 알려진 정적 SQL 조작을 직접 지원합니다. 런타임까지 특정 SQL문의 일부 또는 전체를 결정할 수 없는 경우 이는 동적 조작입니다. 작업 SQLJ 프로그램에서 동적 SQL 조작을 수행하려면 JDBC를 사용하십시오. ConnectionContext 오브젝트에는 JDBC 연결 오브젝트가 있는데 이 오브젝트는 동적 SQL 조작에 필요한 JDBC문 오브젝트를 작성하는 데 사용할 수 있습니다.

모든 SQLJ ConnectionContext 클래스에는 인수로 JDBC 연결을 취하는 구성자가 있습니다. 이 구성자를 사용하여, JDBC 연결의 인스턴스와 그 기본 데이터베이스 연결을 공유하는 SQLJ 연결 문맥 인스턴스를 작성할 수 있습니다.

모든 SQLJ ConnectionContext 인스턴스에는 JDBC Connection 인스턴스를 리턴하는 `getConnection()` 메소드가 있습니다. 리턴되는 JDBC Connection에서는 SQLJ 연결 문맥과 기본 데이터베이스 연결을 공유합니다. 이를 사용하여 JDBC API에 설명된 동적 SQL 조작을 실행할 수도 있습니다.

세션 공유

위에서 설명한 상호운용성 방법을 사용하면 SQLJ에 사용된 연결 추상과 JDBC에 사용된 추상 사이에서 변환할 수 있습니다. 두 추상은 동일 데이터베이스 세션 즉, 기본 데이터베이스 연결을 공유합니다. 따라서 한 오브젝트의 세션 상태에 영향을 주는 메소드에 대한 호출은 다른 오브젝트에도 반영됩니다. 실제로 영향을 받는 기본 공유 세션이므로 그렇습니다.

JDBC에서는 새로 작성된 연결의 세션 상태에 대한 기본 값을 정의합니다. 대부분의 경우 SQLJ에서는 이런 기본값을 채택합니다. 그러나 새로 작성된 JDBC 연결에 기본적으로 자동 확약 모드가 포함된 반면 SQLJ 연결 문맥의 경우 구성시 자동 확약 모드를 명시적으로 지정해야 합니다.

Java의 연결 자원 관리

연결 문맥 인스턴스의 `close()` 메소드를 호출하면 연관된 JDBC 연결 인스턴스와 기본 데이터베이스 연결이 닫힙니다. 연결 문맥이 다른 연결 문맥 및/또는 JDBC 연결과 기본 데이터베이스 연결을 공유할 수 있으므로 연결 문맥이 닫힐 때 기본

데이터베이스 연결을 닫는 것이 좋지 않을 수 있습니다. 프로그래머가 기본 데이터베이스 연결을 실제로 닫지 않고 명령문 핸들과 같은 연결 문맥으로 유지보수되는 자원을 해제하려는 경우가 있을 수 있습니다. 이 목적을 위해 연결 문맥 클래스에서는 close() 메소드도 지원합니다. 이 메소드는 기본 데이터베이스 연결을 닫을 지 여부를 지시하는 부울값 인수를 취합니다. 즉, 데이터베이스 연결을 닫아야 할 경우 CLOSE_CONNECTION 상수, 유지해야 할 경우 KEEP_CONNECTION 상수를 리턴합니다. 인수를 취하지 않는 close()의 변이 메소드는 close (CLOSE_CONNECTION)를 호출하는 단축 방법입니다.

비사용 메모리 정리(garbage collected)를 실행하기 전에 연결 문맥 인스턴스를 명시적으로 닫지 않은 경우 연결 문맥의 최종 정리 메소드에서 close (KEEP_CONNECTION)를 호출합니다. 따라서 연결 관련 자원을 사용할 수 있는 다른 JDBC 및 SQLJ 오브젝트에 대한 기본 데이터베이스 연결을 유지하는 동안 일반 비사용 메모리 정리 프로세스에서 이 자원을 재생할 수 있습니다. 연결을 사용하는 다른 JDBC 또는 SQLJ 오브젝트가 없을 경우 데이터베이스 연결은 닫히고 비사용 메모리 정리 프로세스에서 재생합니다.

SQLJ 연결 문맥 오브젝트와 JDBC 연결 오브젝트는 close() 메소드에 응답합니다. SQLJ 프로그램을 작성하면 연결 문맥 오브젝트에서만 close() 메소드를 호출하는 것으로 충분합니다. 연결 문맥을 닫으면 그와 연관된 JDBC 연결도 닫히기 때문입니다. 그러나 연결 문맥의 getConnection() 메소드에서 리턴한 JDBC 연결만 닫는 것으로는 충분하지 않습니다. JDBC 연결의 close() 메소드로 인해 포함 중인 연결 문맥이 닫히지 않으므로 연결 문맥에서 유지하는 자원이 비사용 메모리 정리 프로세스로 정리되기 전까지 해제되지 않기 때문입니다.

연결 문맥 인스턴스에서 close() 메소드의 변이가 호출될 경우 연결 문맥의 isClosed() 메소드는 true를 리턴합니다. isClosed()에서 true를 리턴할 경우 close() 호출은 아무런 영향도 주지 않으며 또 다른 메소드 호출은 정의되지 않습니다.

제22장 Perl 프로그래밍

Perl 프로그래밍 고려사항 755
Perl 제한사항 755
Perl을 사용한 데이터베이스 연결 756
Perl에서 결과 페치 756
Perl의 매개변수 표시문자 757
Perl의 SQLSTATE 및 SQLCODE 변수 758
Perl DB2 응용프로그램 예 758

Perl 프로그래밍 고려사항

Perl은 여러 운영 체제에서 자유롭게 사용 가능한 인기 있는 프로그래밍 언어입니다. http://www.ibm.com/software/data/db2/perl에서 사용할 수 있는 DBD::DB2 드라이버를 http://www.perl.com에서 사용할 수 있는 Perl (DBI)(Database Interface) 모듈과 함께 사용하면, Perl을 사용하여 DB2 응용프로그램을 작성할 수 있습니다.

Perl은 해석된 언어이고 Perl DBI 모듈은 동적 SQL을 사용하므로 Perl은 DB2 응용프로그램의 프로토타입을 신속하게 작성하고 수정할 수 있는 이상적인 언어입니다. Perl DBI 모듈은 Perl 프로토타입을 CLI 및 JDBC로 쉽게 옮길 수 있게 만들어 주는 CLI 및 JDBC 인터페이스와 꽤 비슷합니다.

대부분의 데이터베이스 벤더는 Perl DBI 모듈에 대해 데이터베이스 드라이버를 제공하는 데, 사용자는 Perl을 사용하여 여러 데이터베이스 서버에서 데이터를 액세스하는 응용프로그램을 작성할 수도 있습니다. 예를 들어, DBD::Oracle 데이터베이스 드라이버를 사용하여 Oracle 데이터베이스에 연결하고 Oracle 데이터베이스에서 데이터를 페치하며 DBD::DB2 데이터베이스 드라이버를 사용하여 DB2 데이터베이스에 데이터를 삽입할 수 있는 Perl DB2 응용프로그램을 작성할 수 있습니다.

Perl 제한사항

Perl DBI 모듈은 동적 SQL만 지원합니다. 명령문을 여러 번 실행해야 하는 경우 `prepare` 호출을 실행하여 명령문을 준비함으로써 Perl DB2 응용프로그램의 성능을 개선할 수 있습니다.

© Copyright IBM Corp. 1993, 2000

워크스테이션에 설치하는 DBD::DB2 드라이버의 버전 제한사항에 대한 현재 정보는 DBD::DB2 드라이버 패키지에 있는 CAVEATS 파일을 참조하십시오.

Perl을 사용한 데이터베이스 연결

Perl을 사용하여 DB2 모듈을 로드하려면 다음 행을 DB2 응용프로그램에 포함시켜야 합니다.

```
use DBI;
```

DBI 모듈은 다음 구문으로 DBI->connect 명령문을 사용하여 *데이터베이스 핸들*을 작성할 때 DBD::DB2 드라이버를 자동으로 로드합니다.

```
my $dbhandle = DBI->connect('dbi:DB2:dbalias', $userID, $password);
```

여기서,

$dbhandle
: 연결 명령문으로 리턴된 데이터베이스 핸들을 나타냅니다.

dbalias
: DB2 데이터베이스 디렉토리에서 카탈로그로 만들어진 DB2 별명을 나타냅니다.

$userID
: 데이터베이스에 연결할 때 사용된 사용자 ID를 나타냅니다.

$password
: 데이터베이스 연결할 때 사용된 사용자 ID의 암호를 나타냅니다.

Perl에서 결과 페치

Perl DBI 모듈은 동적 SQL만 지원하므로 Perl DB2 응용프로그램에서 호스트 변수를 사용하지 않습니다. SQL 조회에서 결과를 리턴하려면 다음 단계를 수행하십시오.

단계 1. 『Perl을 사용한 데이터베이스 연결』에서 설명된 대로 데이터베이스 핸들을 작성합니다.

단계 2. 데이터베이스 핸들에서 명령문 핸들을 작성합니다. 예를 들어, SQL문을 통해 prepare를 문자열 인수로 호출하여 다음 Perl문에서와 같이 데이터베이스 핸들에서 명령문 핸들 *$sth*를 리턴할 수 있습니다.

```
my $sth = $dbhandle->prepare(
   'SELECT firstnme, lastname
      FROM employee '
   );
```

단계 3. 명령문 핸들에서 execute를 호출하여 SQL문을 실행하십시오. 성공적인 execute 호출은 결과 세트를 명령문 핸들과 연관시킵니다. 예를 들어, 다음 Perl문을 사용하여 이전 예제에서 준비된 명령문을 실행할 수 있습니다.

```
#Note: $rc represents the return code for the execute call
my $rc = $sth->execute();
```

단계 4. fetchrow()에 대한 호출로 명령문 핸들에 연관된 결과 세트에서 행을 페치하십시오. Perl DBI는 행을 컬럼당 값이 하나인 배열로 리턴합니다. 예를 들어, 다음 Perl문을 사용하여 이전 예제의 명령문 핸들에서 모든 행을 리턴할 수 있습니다.

```
while (($firstnme, $lastname) = $sth->fetchrow()) {
   print "$firstnme $lastname\n";
}
```

Perl의 매개변수 표시문자

지정된 필드에 대해 여러 입력 값을 사용하여 준비된 명령문을 실행하려면 Perl DBI 모듈은 사용자가 매개변수 표시문자를 사용하여 명령문을 준비하고 실행할 수 있게 합니다. SQl문에 매개변수 표시문자를 포함시키려면 물음표(?) 문자를 사용하십시오.

다음 Perl 코드는 SELECT문의 WHERE절에 대해 매개변수 표시문자를 승인하는 명령문 핸들을 작성합니다. 그러면 코드는 매개변수 표시문자를 바꾸기 위해 입력 값 25000과 35000을 두번 사용하여 명령문을 실행합니다.

```
my $sth = $dbhandle->prepare(
   'SELECT firstnme, lastname
      FROM employee
      WHERE salary > ?'
   );
```

```
my $rc = $sth->execute(25000);
```

⋮

```
my $rc = $sth->execute(35000);
```

Perl의 SQLSTATE 및 SQLCODE 변수

Perl DBI 데이터베이스 핸들 또는 명령문 핸들에 연관된 SQLSTATE를 리턴하려면 `state` 메소드를 호출하십시오. 예를 들어, 데이터베이스 핸들 $dbhandle에 연관된 SQLSTATE를 리턴하려면 다음 Perl문을 응용프로그램에 포함시켜야 합니다.

```
my $sqlstate = $dbhandle->state;
```

Perl DBI 데이터베이스 핸들 또는 명령문 핸들에 연관된 SQLCODE를 리턴하려면 `err` 메소드를 호출합니다. Perl DBI 데이터베이스 핸들 또는 명령문 핸들에 연관된 SQLCODE의 메시지를 리턴하려면 `errstr` 메소드를 호출합니다. 예를 들어, 데이터베이스 핸들 $dbhandle에 연관된 SQLCODE를 리턴하려면 다음 Perl문을 응용프로그램에 포함시켜야 합니다.

```
my $sqlcode = $dbhandle->err;
```

Perl DB2 응용프로그램 예

```
#!/usr/bin/perl
use DBI;

my $database='dbi:DB2:sample';
my $user='';
my $password='';

my $dbh = DBI->connect($database, $user, $password)
   or die "Can't connect to $database: $DBI::errstr";

my $sth = $dbh->prepare(
   q{ SELECT firstnme, lastname
      FROM employee }
   )
   or die "Can't prepare statement: $DBI::errstr";
```

```
my $rc = $sth->execute
   or die "Can't execute statement: $DBI::errstr";

print "Query will return $sth->{NUM_OF_FIELDS} fields.\n\n";
print "$sth->{NAME}->[0]: $sth->{NAME}->[1]\n";

while (($firstnme, $lastname) = $sth->fetchrow()) {
   print "$firstnme: $lastname\n";
}

# check for problems which may have terminated the fetch early
warn $DBI::errstr if $DBI::err;

$sth->finish;
```

제23장 COBOL 프로그래밍

COBOL에 대한 프로그래밍 고려사항 . . . 761
COBOL에서 언어 제한사항 761
COBOL에 대한 입력과 출력 파일 761
COBOL의 Include 파일 762
COBOL에서의 Embedding SQL문. . . . 765
COBOL에서 호스트 변수 768
COBOL에서 오스트 변수 명명 768
호스트 변수 선언 769
COBOL에서 표시기 변수 773
COBOL에서 LOB 선언 773
COBOL에서 LOB LOB 위치 지정자 선언 774
COBOL에서 파일 참조 선언 775
COBOL에서 호스트 구조 유지 776
COBOL에서 표시기 테이블 778
COBOL 그룹 데이터 항목에서 REDEFINES 사용 779
BINARY/COMP-4 COBOL 데이터 유형 사용 780
COBOL에서 지원되는 SQL 데이터 유형 780
COBOL에서 FOR BIT DATA 784
COBOL에서 SQLSTATE 및 SQLCODE 변수 785
COBOL에 대한 일본어 또는 대만어 EUC 및 UCS-2 고려사항 785
오브젝트 지향 COBOL 786

COBOL에 대한 프로그래밍 고려사항

특수 호스트 언어 프로그래밍 고려사항은 다음 페이지에서 다룹니다. 언어 제한사항, 호스트 언어 고유의 include 파일, Embedded SQL문, 호스트 변수 및 호스트 변수에 대해 지원되는 데이터 유형에 대한 정보가 포함됩니다.

COBOL에서 언어 제한사항

모든 API 포인터는 4 바이트입니다. API 호출시 값 매개변수로 사용되는 모든 정수 변수는 USAGE COMP-5절로 선언해야 합니다.

COBOL에 대한 입력과 출력 파일

기본적으로 입력 파일의 확장자는 .sqb이지만 TARGET 사전 처리 컴파일 옵션 TARGET ANSI_COBOL, TARGET IBMCOB, TARGET MFCOB 또는 TARGET MFCOB16을 사용할 경우 입력 파일의 확장자를 선호하는 확장자로 지정할 수 있습니다.

© Copyright IBM Corp. 1993, 2000

기본적으로 출력 파일의 확장자는 .cbl이지만 OUTPUT 사전 처리 컴파일 옵션을 사용하여 출력 수정 소스 파일의 새로운 이름과 경로를 지정할 수 있습니다.

COBOL의 Include 파일

COBOL용 호스트 언어 고유의 include 파일의 확장자는 .cbl입니다. IBM COBOL 컴파일러의 "System/390 호스트 데이터 유형 지원" 기능을 사용할 경우 사용자 응용프로그램용 DB2 include 파일은 다음 디렉토리에 있습니다.

```
$HOME/sqllib/include/cobol_i
```

지정된 스크립트 파일로 DB2 샘플 프로그램을 빌드할 경우 스크립트 파일에 지정된 include 파일 경로를 cobol_a 디렉토리가 아닌 cobol_i 디렉토리로 변경해야 합니다.

IBM COBOL 컴파일러의 "System/390 호스트 데이터 유형 지원" 기능을 **사용하지 않**거나 이 컴파일러의 이전 버전을 사용할 경우 사용자 응용프로그램용 DB2 include 파일은 다음 디렉토리에 있습니다.

```
$HOME/sqllib/include/cobol_a
```

사용자의 응용프로그램에 사용할 include 파일은 아래에서 설명합니다.

SQL (sql.cbl) 이 파일에는 바인더, 사전 처리 컴파일러에 대한 언어 특정 프로토타입과 오류 메시지 API가 있습니다. 또한 시스템 상수를 정의합니다.

SQLAPREP (sqlaprep.cbl)
이 파일에는 사용자의 사전 처리 컴파일러를 작성하기 위해 요구되는 정의가 있습니다.

SQLCA (sqlca.cbl)
이 파일은 SQLCA 구조를 정의합니다. SQLCA에는 SQL문과 API 호출의 실행에 관한 오류 정보와 함께 응용프로그램을 제공할 데이터베이스 관리 프로그램에 의해 사용되는 변수가 있습니다.

SQLCA_92 (sqlca_92.cbl)
이 파일에는 SQL 통신 영역(SQLCA) 구조의 FIPS SQL92 Entry Level 순응 버전이 있습니다. FIPS SQL92 Entry Level 표준을

따르는 DB2 응용프로그램을 작성할 때 이 파일은 sqlca.cbl 파일 대신 포함되어야 합니다. LANGLEVEL 사전 처리 컴파일러 옵션을 SQL92E로 설정하면 DB2 사전 처리 컴파일러에서 sqlca_92.cbl 파일을 자동으로 포함합니다.

SQLCODES (sqlcodes.cbl)

이 파일은 SQLCA 구조의 SQLCODE에 대한 상수를 정의합니다.

SQLDA (sqlda.cbl)

이 파일은 SQLDA 구조를 정의합니다. SQLDA는 응용프로그램과 데이터베이스 관리 프로그램 사이에서 데이터를 전달하기 위해 사용됩니다.

SQLEAU (sqleau.cbl)

이 파일에는 DB2 보안 감사 API에 요구되는 상수 및 구조 정의가 있습니다. 이러한 API를 사용하면 프로그램에서 이 파일을 포함시킬 필요가 있습니다. 또한 이 파일에는 감사 추적 레코드의 필드에 대한 상수 및 키워드 값 정의가 포함됩니다. 외부 또는 벤더 감사 추적 추출 프로그램에서 이런 정의를 사용할 수 있습니다.

SQLENV (sqlenv.cbl)

이 파일은 데이터베이스 환경 API용 언어 특정 호출과 그러한 인터페이스에 대한 구조, 상수 및 리턴 코드를 정의합니다.

SQLETSD (sqletsd.cbl)

이 파일에서는 데이터베이스 API 작성, sqlgcrea로 전달되는 테이블 공간 설명자 구조, SQLETSDESC를 정의합니다.

SQLE819A (sqle819a.cbl)

데이터베이스의 코드 페이지가 819(ISO Latin-1)일 경우, 이 순서는 호스트 CCSID 500(EBCDIC 국제어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLE819B (sqle819b.cbl)

데이터베이스의 코드 페이지가 819(ISO Latin-1)일 경우, 이 순

서는 호스트 CCSID 037(EBCDIC 미국 영어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLE850A (sqle850a.cbl)

데이터베이스의 코드 페이지가 850(ASCII Latin-1)일 경우, 이 순서는 호스트 CCSID 500(EBCDIC 국제어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLE850B (sqle850b.cbl)

데이터베이스의 코드 페이지가 850(ASCII Latin-1)일 경우, 이 순서는 호스트 CCSID 037(EBCDIC 미국 영어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLE932A (sqle932a.cbl)

데이터베이스의 코드 페이지가 932(ASCII Japanese)일 경우, 이 순서는 호스트 CCSID 5035(EBCDIC 일본어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLE932B (sqle932b.cbl)

데이터베이스의 코드 페이지가 932(ASCII Japanese)일 경우, 이 순서는 호스트 CCSID 5026(EBCDIC 일본어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQL1252A (sql1252a.cbl)

데이터베이스의 코드 페이지가 1252(Windows Latin-1)일 경우, 이 순서는 호스트 CCSID 500(EBCDIC 국제어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQL1252B (sql1252b.cbl)

데이터베이스의 코드 페이지가 1252(Windows Latin-1)일 경우, 이 순서는 호스트 CCSID 037(EBCDIC 미국 영어) 2진 집합에

따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLMON (sqlmon.cbl)

이 파일은 데이터베이스 시스템 모니터 API용 언어 특정 호출과 그러한 인터페이스에 대한 구조, 상수 및 리턴 코드를 정의합니다.

SQLMONCT (sqlmonct.cbl)

이 파일에는 데이터베이스 시스템 모니터 API를 호출하기 위해 필요한 상수 정의와 지역 데이터 구조 정의가 있습니다.

SQLSTATE (sqlstate.cbl)

이 파일은 SQLCA 구조의 SQLSTATE 필드에 대한 상수를 정의합니다.

SQLUTBCQ (sqlutbcq.cbl)

이 파일에서는 테이블 공간 컨테이너 조회 데이터 구조 SQLB-TBSCONTQRY-DATA를 정의합니다. 이 데이터 구조는 테이블 공간 컨테이너 조회 API, sqlgstsc, sqlgftcq 및 sqlgtcq에 사용됩니다.

SQLUTBSQ (sqlutbsq.cbl)

이 파일에서는 테이블 공간 조회 데이터 구조 SQLB-TBSQRY-DATA를 정의합니다. 이 데이터 구조는 테이블 공간 조회 API, sqlgstsq, sqlgftsq 및 sqlgtsq에 사용됩니다.

SQLUTIL (sqlutil.cbl)

이 파일은 유틸리티 API용 언어 특정 호출과 그러한 인터페이스에 요구되는 구조, 상수 및 코드를 정의합니다.

COBOL에서의 Embedding SQL문

Embedded SQL문은 다음 세 구성요소로 구성됩니다.

구성요소	올바른 COBOL 구문
키워드 쌍	EXEC SQL
명령문 문자열	유효한 SQL문

명령문 종료 표시자 END-EXEC.

예를 들면 다음과 같습니다.

```
EXEC SQL SELECT col INTO :hostvar FROM table END-EXEC.
```

다음 규칙이 Embedded SQL문에 적용됩니다.

- 실행 가능 SQL문은 PROCEDURE DIVISION에 지정해야 합니다. COBOL 문과 같은 단락 이름을 SQL문 앞에 지정할 수 있습니다.
- SQL문은 영역 A(컬럼 8 - 11) 또는 영역 B(컬럼 12 - 72)에서 시작할 수 있습니다.
- EXEC SQL로 각 SQL문을 시작하고 END-EXEC로 종료하십시오. SQL 사전 처리 컴파일러는 각 SQL문을 수정 소스 파일의 주석으로 포함합니다.
- SQL문 종료 표시자를 사용해야 합니다. 종료 표시자를 사용하지 않으면 사전 처리 컴파일러가 응용프로그램에서 다음 종료 표시자가 나타날 때까지 계속 실행됩니다. 따라서 종료 오류가 발생할 수 있습니다.
- Embedded SQL문의 일부인 모든 행에서 SQL 주석이 허용됩니다. 동적으로 실행되는 명령문에서는 이런 주석이 허용되지 않습니다. SQL 주석의 형식은 이중 대시(--) 다음에 0개 이상의 문자로 구성되며 행의 끝으로 종료됩니다. COBOL 언어의 일부인 것처럼 보여 컴파일 오류가 발생되므로 SQL문 종료 표시자 뒤에 SQL문을 지정하지 마십시오.
- COBOL 주석은 Embedded SQL문 안의 *거의* 모든 위치에서 허용됩니다. 예외는 다음과 같습니다.
  - EXEC와 SQL 사이에서는 주석이 허용되지 않습니다.
  - 동적으로 실행되는 명령문에서는 주석이 허용되지 않습니다.
- SQL문은 COBOL 언어와 동일한 행 지속 규칙을 따릅니다. 행 사이의 EXEC SQL 키워드 쌍을 분리하지 마십시오.
- COBOL COPY문을 사용하여 SQL문이 있는 파일을 포함하지 마십시오. 모듈을 컴파일하기 전에 SQL문이 사전 처리 컴파일됩니다. 사전 처리 컴파일러는 COBOL COPY문을 무시합니다. 대신 SQL INCLUDE문을 사용하여 해당 파일을 포함하십시오.

INCLUDE 파일을 찾기 위해 DB2 COBOL 사전 처리 컴파일러에서는 먼저 현재 디렉토리를 검색한 다음 DB2INCLUDE 환경 변수로 지정된 디렉토리를 검색합니다. 다음 예를 고려하십시오.

– `EXEC SQL INCLUDE payroll END-EXEC.`

  위와 같이 INCLUDE문에서 지정한 파일을 인용 부호로 닫지 않은 경우 사전 처리 컴파일러는 자신이 찾은 각 디렉토리에서 `payroll.sqb`, `payroll.cpy`, `payroll.cbl` 순으로 검색합니다.

– `EXEC SQL INCLUDE 'pay/payroll.cbl' END-EXEC.`

  위와 같이 파일 이름을 인용 부호로 닫을 경우 확장자는 이름에 추가되지 않습니다.

  인용 부호 안의 파일 이름에 절대 경로가 없을 경우 INCLUDE 파일 이름에 지정된 경로에 따라 DB2INCLUDE의 내용을 사용하여 파일을 검색합니다. 예를 들어, AIX용 DB2를 사용하여, DB2INCLUDE를 '`/disk2:myfiles/cobol`'로 설정하면 사전 처리 컴파일러에서 '`./pay/payroll.cbl`', '`/disk2/pay/payroll.cbl`', '`./myfiles/cobol/pay/payroll.cbl`' 순으로 검색합니다. 실제로 파일이 있는 경로가 사전 처리 컴파일러 메시지에 표시됩니다. OS/2 및 Windows 플랫폼의 경우 위 예에서 정방향 슬래시를 역슬래시(\)로 대체하십시오.

주: DB2 명령행 처리기에서 DB2INCLUDE의 설정을 캐쉬화합니다. CLP 명령을 발행한 후 DB2INCLUDE의 설정을 변경하려면 TERMINATE 명령을 입력한 다음 데이터베이스에 다시 연결하고 평소와 마찬가지로 사전 처리 컴파일하십시오.

- 문자열 상수가 다음 행으로 연속되게 하려면 연속되는 행의 컬럼 7에 '-'를 포함하고 컬럼 12 이상에 문자열 분리문자를 포함해야 합니다.
- SQL 산술 연산자는 공백으로 분리해야 합니다.
- 전체 행 COBOL 주석은 SQL문 안을 포함하여 프로그램 내 어느 위치에나 나타날 수 있습니다.
- SQL문에서 호스트 변수를 참조할 때 선언된 것과 동일한 호스트 변수를 사용하십시오.
- 행의 끝(eol)과 TAB 문자와 같은 공백 문자 대체는 다음과 같이 발생합니다.

- SQL문 안이지만 외부 인용 부호 외부에 나타날 경우 행의 끝과 TAB은 단일 공백으로 대체됩니다.
- 인용 부호 안에서 나타날 때 COBOL 프로그램에서 문자열이 제대로 연속되지 않을 경우 문자열이 행의 끝 문자가 사라집니다. TAB은 수정되지 않습니다.

행의 끝과 TAB에 사용되는 실제 문자는 플랫폼에 따라 달라진다는 점에 주의하십시오. 예를 들어, OS/2에서는 행의 끝에 대해 캐리지 리턴이나 라인 피드를 사용하지만 UNIX 기반 시스템에서는 라인 피드만 사용합니다.

COBOL에서 호스트 변수

호스트 변수는 SQL문에서 참조되는 COBOL 언어 변수입니다. 이런 변수를 사용하여 응용프로그램에서 데이터베이스 관리 프로그램으로 입력 데이터를 전달하고 데이터베이스 관리 프로그램으로부터 출력 데이터를 수신할 수 있습니다. 응용프로그램을 사전 처리 컴파일하면 컴파일러에서 다른 COBOL 변수와 마찬가지로 호스트 변수를 사용합니다. 호스트 변수를 명명하고, 선언하고, 사용할 때 아래에 기술된 규칙을 따르십시오.

COBOL에서 오스트 변수 명명

SQL 사전 처리 컴파일러에서 선언된 이름으로 호스트 변수를 식별합니다. 다음 규칙이 적용됩니다.

- 최대 255자까지 변수 이름을 지정하십시오.
- 시스템용으로 예약된 SQL, sql, DB2 또는 db2 이외의 접두부로 호스트 변수를 시작하십시오.
- 아래 기술된 선언 구문을 사용하는 FILLER 항목은 그룹 호스트 변수 선언에서 허용되며 사전 처리 컴파일러에서는 이를 무시합니다. 그러나 SQL DECLARE절에서 FILLER를 한번 이상 사용할 경우 사전 처리 컴파일러는 실패합니다. VARCHAR, LONG VARCHAR, VARGRAPHIC 또는 LONG VARGRAPHIC 선언에는 FILLER 항목을 포함하지 못할 수 있습니다.
- 호스트 변수 이름에 하이픈을 사용할 수 있습니다.

  SQL은 양 옆에 공백이 있는 하이픈을 뺄셈 연산자로 해석합니다. 따라서 호스트 변수 이름에서 공백 없이 하이픈을 사용하십시오.

- 호스트 변수 선언에서 REDEFINES절이 허용됩니다.
- 호스트 변수 선언 절에서 레벨 88 선언이 허용되지만 무시됩니다.

호스트 변수 선언

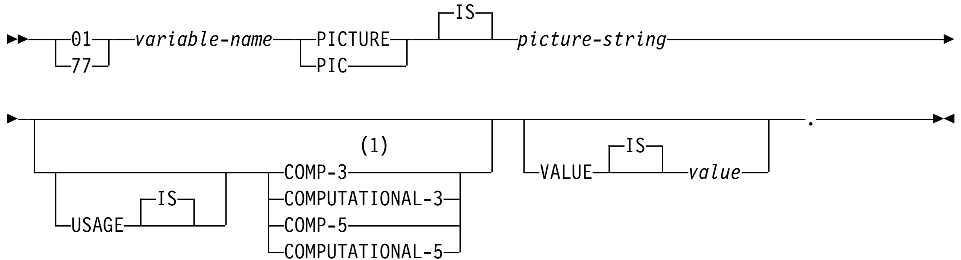
SQL 선언 절은 호스트 변수 선언을 식별하는 데 사용해야 합니다. 이 절은 후속 SQL문에서 참조할 수 있는 호스트 변수를 후속 사전 처리 컴파일러에 알려 줍니다.

구조화 유형에 대해 호스트 변수를 선언하는 것에 대한 정보는 381 페이지의 『구조화 유형 호스트 변수 선언』을 참조하십시오.

COBOL 사전 처리 컴파일러만이 유효한 COBOL 선언의 부속 집합을 인식합니다.

COBOL에서 숫자 호스트 변수의 구문은 숫자 호스트 변수의 구문을 나타냅니다.

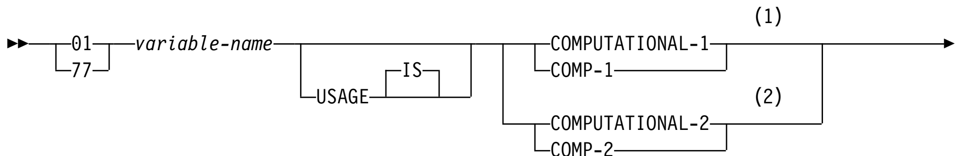
COBOL에서 숫자 호스트 변수의 구문



주:

1 COMP-3의 대체는 압축 십진수입니다.

부동 소수점



주:

1 REAL (SQLTYPE 480), 길이 4

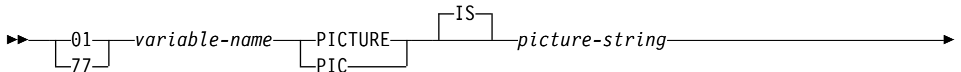
2 DOUBLE (SQLTYPE 480), 길이 8

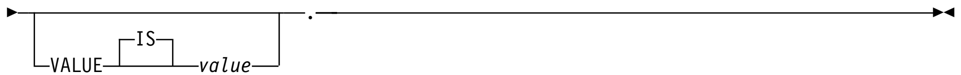
숫자 호스트 변수 고려사항:

1. *Picture-string*의 양식은 다음 중 하나와 같아야 합니다.
   - S9(m)V9(n)
   - S9(m)V
   - S9(m)
2. 9는 확장될 수 있습니다. 예를 들면, S9(3)″ 대신 ″S999″가 들어갈 수 있습니다.
3. *m*과 *n*은 양의 정수여야 합니다.

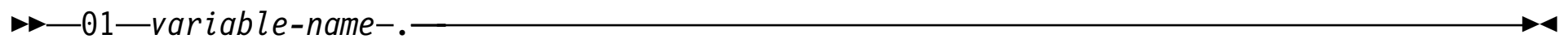
COBOL에서 문자 호스트 변수의 구문: 고정 길이는 문자 호스트 변수의 구문을 나타냅니다.

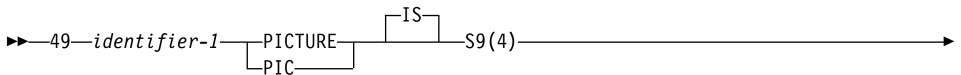
COBOL에서 문자 호스트 변수의 구문: 고정 길이

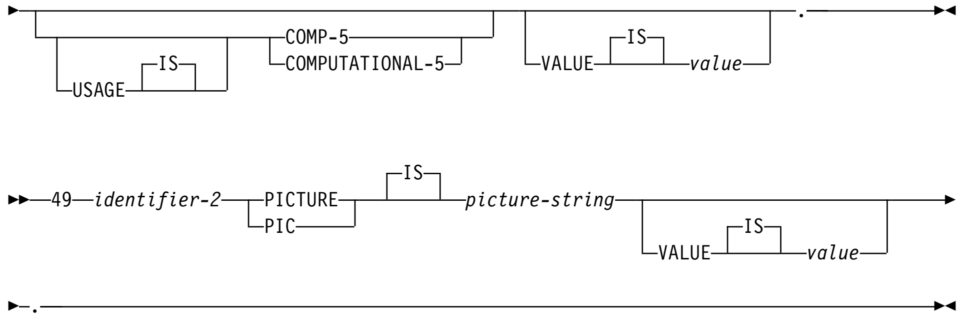




변수 길이







문자 호스트 변수 고려사항:

1. *Picture-string*의 양식은 *X(m)*여야 합니다. 또는 X는 확장될 수 있습니다. 예를 들면, "X(3)" 대신 "XXX"가 들어갈 수 있습니다.
2. 고정 길이 문자열의 경우 *m*의 범위는 1 - 254입니다.
3. 가변 길이 문자열의 경우 *m*의 범위는 1 - 32 700입니다.
4. *m*이 32 672보다 클 경우 호스트 변수는 LONG VARCHAR 문자열로 처리되며 그 사용은 제한될 수 있습니다.
5. PICTURE절에서 X와 9를 picture character로 사용하십시오. 다른 문자는 허용되지 않습니다.
6. 가변 길이 문자열은 길이 항목과 값 항목으로 구성됩니다. 길이 항목과 문자열 항목에 수용 가능한 COBOL 이름을 사용할 수 있습니다. 그러나 SQL문의 집합적인 이름으로 가변 길이 문자열을 참조하십시오.
7. 아래와 같은 CONNECT문에서 COBOL 문자열 호스트 변수 `dbname`과 `userid` 뒤에 공백이 포함되지만 처리 전에 제거됩니다.

   ```
   EXEC SQL CONNECT TO :dbname USER :userid USING :p-word
   END-EXEC.
   ```

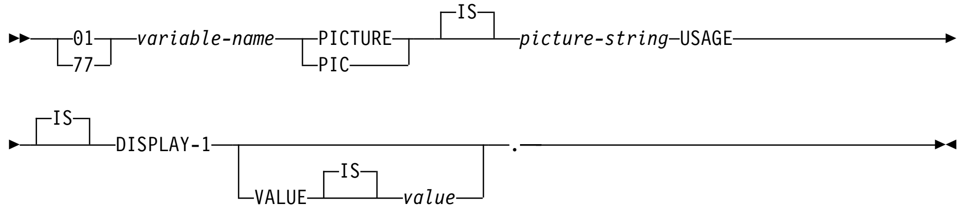
   그러나 암호에서 공백이 중요할 수 있으므로, 응용프로그램에서 다음과 같은 CONNECT문의 중요한 암호 길이를 명시적으로 지시할 수 있도록 `p-word` 호스트 변수를 VARCHAR 데이터 항목으로 선언해야 합니다.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   01 dbname PIC X(8).
   01 userid PIC X(8).
   01 p-word.
```

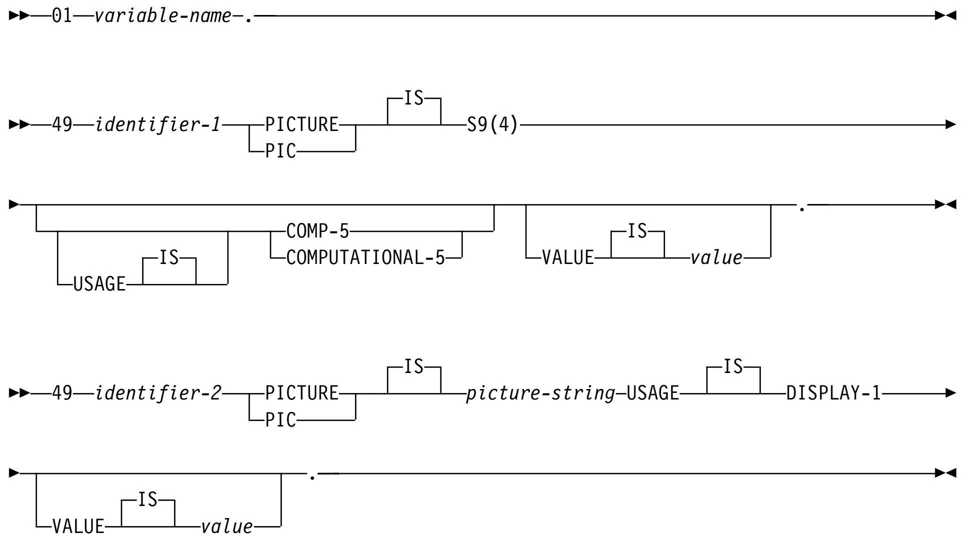
```
         49 L PIC S9(4) COMP-5.
         49 D PIC X(18).
   EXEC SQL END DECLARE SECTION END-EXEC.
      PROCEDURE DIVISION.
         MOVE "sample" TO dbname.
         MOVE "userid" TO userid.
         MOVE "password" TO D OF p-word.
         MOVE 8          TO L of p-word.
      EXEC SQL CONNECT TO :dbname USER :userid USING :p-word
      END-EXEC.
```

COBOL에서 그래픽 호스트 변수의 구문: 고정 길이는 그래픽 호스트 변수의 구문을 나타냅니다.

COBOL에서 그래픽 호스트 변수의 구문: 고정 길이



변수 길이



그래픽 호스트 변수 고려사항:

1. *Picture-string*의 양식은 *G(m)*여야 합니다. 또는 G는 확장될 수 있습니다. 예를 들면, "G(3)" 대신 "GGG"가 들어갈 수 있습니다.
2. 고정 길이 문자열의 경우 *m*의 범위는 1 - 127입니다.
3. 가변 길이 문자열의 경우 *m*의 범위는 1 - 16 350입니다.
4. *m*이 16 336보다 클 경우 호스트 변수는 LONG VARGRAPHIC 문자열로 처리되며 그 사용은 제한될 수 있습니다.

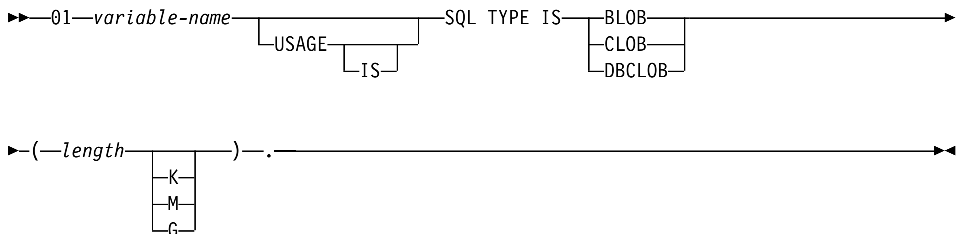
COBOL에서 표시기 변수

표시기 변수는 `PIC S9(4) COMP-5` 데이터 유형으로 선언해야 합니다.

COBOL에서 LOB 선언

COBOL에서 LOB 호스트 변수의 구문은 COBOL에서 대형 오브젝트(LOB) 호스트 변수를 선언하는 구문을 나타냅니다.

COBOL에서 LOB 호스트 변수의 구문



LOB 호스트 변수 고려사항:

1. BLOB and CLOB의 경우 1 <= lob-length <= 2 147 483 647
2. DBCLOB의 경우 1 <= lob-length <= 1 073 741 823.
3. SQL TYPE IS, BLOB, CLOB, DBCLOB, K, M, G는 대문자, 소문자 또는 대소문자 혼합으로 나타날 수 있습니다.
4. LOB 선언에서 초기화는 허용되지 않습니다.
5. 사전 처리 컴파일러 생성 코드에서 호스트 변수 이름 앞에는 LENGTH와 DATA가 지정됩니다.

BLOB 예:

선언:

```
01 MY-BLOB USAGE IS SQL TYPE IS BLOB(2M).
```

그 결과 다음 구조가 생성됩니다.

```
01 MY-BLOB.
   49 MY-BLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-BLOB-DATA PIC X(2097152).
```

CLOB 예:

선언:

```
01 MY-CLOB USAGE IS SQL TYPE IS CLOB(125M).
```

그 결과 다음 구조가 생성됩니다.

```
01 MY-CLOB.
   49 MY-CLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-CLOB-DATA PIC X(131072000).
```

DBCLOB 예:

선언:

```
     01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(30000).
```

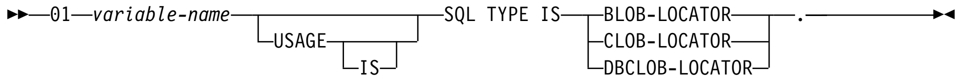
그 결과 다음 구조가 생성됩니다.

```
01 MY-DBCLOB.
   49 MY-DBCLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-DBCLOB-DATA PIC G(30000) DISPLAY-1.
```

COBOL에서 LOB LOB 위치 지정자 선언

COBOL에서 LOB 위치 지정자 호스트 변수의 구문은 COBOL에서 대형 오브젝트(LOB) 위치 지정자 호스트 변수를 선언하는 구문을 나타냅니다.

COBOL에서 LOB 위치 지정자 호스트 변수의 구문



LOB 위치 지정자 호스트 변수 고려사항:

1. SQL TYPE IS, BLOB-LOCATOR, CLOB-LOCATOR, DBCLOB-LOCATOR은 대문자, 소문자 또는 대소문자 혼합으로 나타날 수 있습니다.
2. 위치 지정자의 초기화는 허용되지 않습니다.

BLOB 위치 지정자 예(다른 LOB 위치 지정자 유형은 비슷):

선언:

```
01 MY-LOCATOR USAGE SQL TYPE IS BLOB-LOCATOR.
```

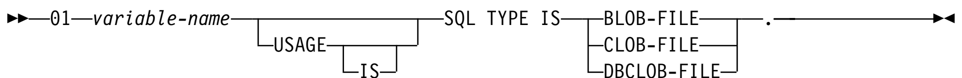
그 결과 다음 선언이 생성됩니다.

```
01 MY-LOCATOR PIC S9(9) COMP-5.
```

COBOL에서 파일 참조 선언

COBOL에서 파일 참조 호스트 변수의 구문은 COBOL에서 파일 참조 호스트 변수를 선언하는 구문을 나타냅니다.

COBOL에서 파일 참조 호스트 변수의 구문



- SQL TYPE IS, BLOB-FILE, CLOB-FILE, DBCLOB-FILE은 대문자, 소문자 또는 대소문자 혼합으로 나타날 수 있습니다.

BLOB 파일 참조 예(다른 LOB 유형은 비슷):

선언:

```
01 MY-FILE USAGE IS SQL TYPE IS BLOB-FILE.
```

그 결과 다음 선언이 생성됩니다.

```
01 MY-FILE.
   49 MY-FILE-NAME-LENGTH PIC S9(9) COMP-5.
   49 MY-FILE-DATA-LENGTH PIC S9(9) COMP-5.
   49 MY-FILE-FILE-OPTIONS PIC S9(9) COMP-5.
   49 MY-FILE-NAME PIC X(255).
```

COBOL에서 호스트 구조 유지

COBOL 사전 처리 컴파일러에서는 호스트 변수 선언 절에서의 그룹 데이터 항목에 대한 선언을 지원합니다. 따라서 무엇보다도 SQL문의 기본 데이터 항목 집합을 간편하게 참조할 수 있습니다. 예를 들어, 다음 그룹 데이터 항목을 사용하여 SAMPLE 데이터베이스의 STAFF 테이블에 있는 일부 컬럼에 액세스할 수 있습니다.

```
01 staff-record.
    05 staff-id       pic s9(4) comp-5.
    05 staff-name.
        49 l          pic s9(4) comp-5.
        49 d          pic x(9).
    05 staff-info.
        10 staff-dept pic s9(4) comp-5.
        10 staff-job  pic x(5).
```

선언 절의 그룹 데이터 항목은 종속 데이터 항목으로 위에서 설명한 유효한 호스트 변수를 포함할 수 있습니다. 여기에는 모든 숫자와 문자 유형, 모든 대형 오브젝트(LOB)가 들어갑니다. 10 레벨까지 그룹 데이터 항목을 중첩할 수 있습니다. 위의 예와 같이 49 레벨의 종속 항목으로 VARCHAR 문자 유형을 선언해야 한다는 점에 주의하십시오. 49 레벨에 있지 않을 경우 VARCHAR은 두 종속 항목을 가진 그룹 데이터 항목으로 처리되며 그룹 데이터 항목 선언 및 사용의 규칙을 따릅니다. 위의 예에서 `staff-info`는 그룹 데이터 항목인 반면 `staff-name`은 VARCHAR입니다. 같은 원칙이 LONG VARCHAR, VARGRAPHIC 및 LONG VARGRAPHIC에 적용됩니다. 02와 49 사이의 레벨에서 그룹 데이터 항목을 선언할 수 있습니다.

그룹 데이터 항목과 그 후속 항목은 다음과 같이 네 가지 방법으로 사용할 수 있습니다.

방법 1.

SQL문에서 전체 그룹을 단일 호스트 변수로 참조할 수 있습니다.

```
EXEC SQL SELECT id, name, dept, job
      INTO :staff-record
  FROM staff WHERE id = 10 END-EXEC.
```

사전 처리 컴파일러에서 staff-record에 대한 참조를 staff-record에서 선언된 모든 후속 항목의 목록으로 변환합니다. 이때 목록은 쉼표로 분리됩니다. 각 기본 항목은 모든 레벨의 그룹 이름으로 규정되므로 다른 항목과의 명명 충돌을 방지합니다. 이는 다음 방법과 일치합니다.

방법 2.

그룹 데이터 항목을 사용하는 두 번째 방법은 다음과 같습니다.

```
EXEC SQL SELECT id, name, dept, job
      INTO
      :staff-record.staff-id,
      :staff-record.staff-name,
      :staff-record.staff-info.staff-dept,
      :staff-record.staff-info.staff-job
  FROM staff WHERE id = 10 END-EXEC.
```

주: staff-id에 대한 참조는 순수 COBOL의 경우와 같이 staff-record의 staff-id가 아닌 staff-record. 접두부를 사용하는 그룹 이름으로 규정됩니다.

위 명령문, staff-record의 종속 항목과 동일한 이름을 가진 호스트 변수가 없다고 가정하면 명시적 그룹 규정을 제거하고 방법 3의 경우와 같이 코딩될 수도 있습니다.

방법 3.

여기서 종속 항목은 특정 그룹 항목으로 규정되지 않고 일반 COBOL 방식으로 참조됩니다.

```
EXEC SQL SELECT id, name, dept, job
      INTO
      :staff-id,
      :staff-name,
      :staff-dept,
      :staff-job
  FROM staff WHERE id = 10 END-EXEC.
```

순수 COBOL에서와 마찬가지로, 지정된 종속 항목을 고유하게 식별할 수 있는 한 사전 처리 컴파일러에서 이 방법을 사용할 수 있습니다. 예를 들어, staff-job이 하나 이상의 그룹에 나타날 경우 사전 처리 컴파일러에서 모호한 참조를 지시하는 오류를 발행합니다.

```
SQL0088N Host variable "staff-job" is ambiguous.
```

방법 4.

모호한 참조를 해결하기 위해 다음과 같이 후속 항목에 대해 부분적인 규정을 사용할 수 있습니다.

```
EXEC SQL SELECT id, name, dept, job
      INTO
      :staff-id,
      :staff-name,
      :staff-info.staff-dept,
      :staff-info.staff-job
  FROM staff WHERE id = 10 END-EXEC.
```

방법 1에서와 같이 그룹 항목에 대한 참조가 쉼표로 분리되는 후속 항목의 목록에 해당하므로 이런 유형의 참조가 오류를 일으킬 수 있는 인스턴스가 있습니다. 예를 들어, 다음과 같습니다.

```
EXEC SQL CONNECT TO :staff-record END-EXEC.
```

여기에서 CONNECT문은 단일 문자 호스트 변수를 기대합니다. 대신 staff-record 그룹 데이터 항목을 지정하면 호스트 변수는 다음과 같은 사전 처리 컴파일 시간 오류를 낳습니다.

```
SQL0087N "staff-record" 호스트 변수는 구조 참조가 허용되지 않을 때
        사용되는 구조입니다.
```

SQL0087N을 발생시키는 그룹 항목의 기타 사용에는 PREPARE, EXECUTE IMMEDIATE, CALL, 표시기 변수 및 SQLDA 참조가 들어갑니다. 한 종속 항목만 포함된 그룹은 위의 방법 2, 3, 4에서와 같이 개별 종속 항목을 참조하는 경우에 허용됩니다.

COBOL에서 표시기 테이블

COBOL 사전 처리 컴파일러에서는 그룹 데이터 항목에 사용하면 편리한 표시기 변수의 테이블 선언을 지원합니다. 이는 다음과 같이 선언됩니다.

```
01 <indicator-table-name>.
    05 <indicator-name> pic s9(4) comp-5
                        occurs <table-size> times.
```

예를 들어, 다음과 같습니다.

```
01 staff-indicator-table.
    05 staff-indicator pic s9(4) comp-5
                       occurs 7 times.
```

이 표시기 테이블은 위 그룹 항목 참조의 첫번째 형식으로 효과적으로 사용할 수 있습니다.

```
EXEC SQL SELECT id, name, dept, job
  INTO :staff-record :staff-indicator
  FROM staff WHERE id = 10 END-EXEC.
```

여기서 사전 처리 컴파일러는 staff-indicator가 표시기 테이블로 선언되었음을 탐지하고, SQL문 처리시 개별 표시기 참조로 확장합니다. staff-indicator(1)는 staff-record의 staff-id와 연관되고 staff-indicator(2)는 staff-record의 staff-name과 연관되는 식입니다.

주: 표시기 테이블의 표시기 항목이 데이터 항목에 있는 종속 항목보다 k개 많을 경우(예를 들어, staff-indicator에 10개의 항목이 있으면 k=6), 표시기 테이블 끝에서 k개의 추가 항목은 무시됩니다. 마찬가지로 표시기 항목이 종속 항목보다 k개 적을 경우 그룹 항목의 마지막 k개의 종속 항목에는 그와 연관된 표시기가 없습니다. **SQL문의 표시기 테이블에 있는 개별 구성요소를 참조할 수 있습니다.**

COBOL 그룹 데이터 항목에서 REDEFINES 사용

호스트 변수 선언시 REDEFINES절을 사용할 수 있습니다. REDEFINES절로 그룹 데이터 항목의 구성원을 선언하고 SQL문에서 그 그룹 데이터 항목을 전체로 참조될 경우 REDEFINES절을 포함한 모든 종속 항목은 확장되지 않습니다. 예를 들면 다음과 같습니다.

```
01 foo.
 10 a pic s9(4) comp-5.
 10 a1 redefines a pic x(2).
 10 b pic x(10).
```

SQL문에서 foo에 대한 참조는 다음과 같습니다.

```
... INTO :foo ...
```

위의 명령문은 다음에 해당합니다.

```
... INTO :foo.a, :foo.b ...
```

즉, REDEFINES절로 선언된 a1 종속 항목은 이런 경우에 자동으로 확장되지 않습니다. a1이 명확한 경우 SQL문에서 REDEFINES절로 종속 항목을 명시적으로 참조할 수 있습니다.

```
... INTO :foo.a1 ...
```

또는

```
... INTO :a1 ...
```

BINARY/COMP-4 COBOL 데이터 유형 사용

목표 COBOL 컴파일러에서 COMP-5 데이터 유형에 해당하는 BINARY, COMP 또는 COMP-4 데이터 유형을 보는 한(또는 보도록 설정할 수 있는 한) 정수 호스트 변수와 표시기가 허용될 때마다 DB2 COBOL 사전 처리 컴파일러에서 BINARY, COMP 및 COMP-4 데이터 유형 사용을 지원합니다. 이 책에서 이런 호스트 변수와 표시기는 COMP-5 유형으로 표시됩니다. COMP, COMP-4, BINARY COMP 및 COMP-5를 동일하게 처리하는 DB2에서 지원하는 목표 컴파일러는 다음과 같습니다.

- AIX용 IBM COBOL Set
- AIX용 Micro Focus COBOL
- OS/2용 IBM COBOL Visual Set(-qbinary(원시) 옵션 집합이 포함된)
- OS/2, Windows NT 및 Windows 95용 IBM VisualAge for COBOL(-qbinary(원시) 옵션 집합이 포함된)

COBOL에서 지원되는 SQL 데이터 유형

사전 정의된 특정 COBOL 데이터 유형은 컬럼 유형에 해당합니다. 이런 COBOL 데이터 유형은 호스트 변수로 선언할 수 있습니다.

표33은 각 컬럼 유형에 해당하는 COBOL 데이터 유형을 나타냅니다. 사전 처리 컴파일러에서 호스트 변수 선언을 찾으면 해당 SQL 유형 값을 판별합니다. 데이터베이스 관리 프로그램에서 이 값을 사용하여 응용프로그램과 자체에서 교환된 데이터를 변환합니다.

호스트 변수에 대한 가능한 데이터 설명이 모두 인식되지는 않습니다. COBOL 데이터 항목은 다음 테이블에서 설명된 데이터 항목과 일치해야 합니다. 다른 데이터 항목을 사용할 경우 오류가 일어날 수 있습니다.

주: 모든 DB2 호스트 언어에서 DATALINK 데이터 유형에 대한 호스트 변수 지원은 없습니다.

표 33. COBOL 선언에 맵핑되는 SQL 데이터 유형

SQL 컬럼 유형[1]	COBOL 데이터 유형	SQL 컬럼 유형 설명
SMALLINT (500 또는 501)	01 name PIC S9(4) COMP-5.	부호있는 16비트 정수
INTEGER (496 또는 497)	01 name PIC S9(9) COMP-5.	부호있는 32비트 정수
BIGINT (492 또는 493)	01 name PIC S9(18) COMP-5.	부호있는 64비트 정수
DECIMAL(*p*,*s*) (484 또는 485)	01 name PIC S9(*m*)V9(*n*) COMP-3.	팩된 10진수
REAL[2] (480 또는 481)	01 name USAGE IS COMP-1.	단정밀 부동 소수점
DOUBLE[3] (480 또는 481)	01 name USAGE IS COMP-2.	배정밀 부동 소수점
CHAR(*n*) (452 또는 453)	01 name PIC X(*n*).	고정 길이 문자열
VARCHAR(*n*) (448 또는 449)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC X(*n*). 1<=*n*<=32 672	가변 길이 문자열
LONG VARCHAR (456 또는 457)	01 name. 49 length PIC S9(4) COMP-5. 49 data PIC X(*n*). 32 673<=*n*<=32 700	긴 가변 길이 문자열

표 33. COBOL 선언에 맵핑되는 SQL 데이터 유형 (계속)

SQL 컬럼 유형[1]	COBOL 데이터 유형	SQL 컬럼 유형 설명
CLOB(*n*) (408 또는 409)	01 MY-CLOB USAGE IS SQL TYPE IS CLOB(n). 1<=*n*<=2 147 483 647	대형 오브젝트(LOB) 가변 길이 문자열
CLOB 위치 지정자 변수[4] (964 또는 965)	01 MY-CLOB-LOCATOR USAGE IS SQL TYPE IS CLOB-LOCATOR.	서버의 CLOB 엔터티를 식별합니다.
CLOB 파일 참조 변수[4] (920 또는 921)	01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.	CLOB 데이터를 포함한 파일의 설명자
BLOB(*n*) (404 또는 405)	01 MY-BLOB USAGE IS SQL TYPE IS BLOB(n). 1<=*n*<=2 147 483 647	대형 오브젝트(LOB) 가변 길이 2진 문자열
BLOB 위치 지정자 변수[4] (960 또는 961)	01 MY-BLOB-LOCATOR USAGE IS SQL TYPE IS BLOB-LOCATOR.	서버의 BLOB 엔터티를 식별합니다.
BLOB 파일 참조 변수[4] (916 또는 917)	01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.	CLOB 데이터를 포함한 파일의 설명자
DATE (384 또는 385)	01 identifier PIC X(10).	10바이트 문자열
TIME (388 또는 389)	01 identifier PIC X(8).	8바이트 문자열
TIMESTAMP (392 또는 393)	01 identifier PIC X(26).	26바이트 문자열
주: 다음 데이터 유형은 DBCS 환경에서만 사용할 수 있습니다.		
GRAPHIC(*n*) (468 또는 469)	01 name PIC G(*n*) DISPLAY-1.	고정 길이 2 바이트 문자열
VARGRAPHIC(*n*) (464 또는 465)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC G(*n*) DISPLAY-1. 1<=*n*<=16 336	2 바이트 문자열 길이 표시기를 포함하는 가변 길이의 2 바이트 문자열
LONG VARGRAPHIC (472 또는 473)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC G(*n*) DISPLAY-1. 16 337<=*n*<=16 350	2 바이트 문자열 길이 표시기를 포함하는 가변 길이의 2 바이트 문자열
DBCLOB(*n*) (412 또는 413)	01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(n). 1<=*n*<=1 073 741 823	4 바이트 문자열 길이 표시기를 포함하는 대형 오브젝트(LOB) 가변 길이의 2 바이트 문자열
DBCLOB 위치 지정자 변수[4] (968 또는 969)	01 MY-DBCLOB-LOCATOR USAGE IS SQL TYPE IS DBCLOB-LOCATOR.	서버의 DBCLOB 엔터티를 식별합니다.

표 33. COBOL 선언에 맵핑되는 SQL 데이터 유형 (계속)

SQL 컬럼 유형[1]	COBOL 데이터 유형	SQL 컬럼 유형 설명
DBCLOB 파일 참조 변수[4] (924 또는 925)	01 MY-DBCLOB-FILE USAGE IS SQL TYPE IS DBCLOB-FILE.	DBCLOB 데이터를 포함한 파일의 설명자

주:

1. **SQL 컬럼 유형** 아래서 첫번째 숫자는 표시기 변수가 제공되지 않은 경우를 나타내며 두 번째 숫자는 표시기 변수가 제공된 경우를 나타냅니다. 표시기 변수는 널(NULL) 값을 나타내거나 절단된 문자열의 길이를 유지하는 데 필요합니다. 이들은 해당 데이터 유형에 대해 SQLDA의 SQLTYPE에 나타나는 값입니다.
2. FLOAT(*n*) 여기서 $0 < n < 25$는 REAL의 동의어입니다. SQLDA에서 REAL과 DOUBLE 사이의 차이점은 길이 값(4 또는 8)입니다.
3. 다음 SQL 유형은 DOUBLE의 동의어입니다.
   - FLOAT
   - FLOAT(*n*) 여기서 $24 < n < 54$
   - DOUBLE PRECISION
4. 이는 컬럼 유형이 아닌 호스트 변수 유형입니다.

다음은 지원되는 각 SQL 데이터 유형에 대해 선언된 호스트 변수를 포함한 샘플 SQL 선언 절입니다.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   *
     01 age        PIC S9(4) COMP-5.
     01 divis      PIC S9(9) COMP-5.
     01 salary     PIC S9(6)V9(3) COMP-3.
     01 bonus      USAGE IS COMP-1.
     01 wage       USAGE IS COMP-2.
     01 nm         PIC X(5).
     01 varchar.
        49 leng    PIC S9(4) COMP-5.
        49 strg    PIC X(14).
     01 longvchar.
        49 len     PIC S9(4) COMP-5.
        49 str     PIC X(6027).
     01 MY-CLOB USAGE IS SQL TYPE IS CLOB(1M).
     01 MY-CLOB-LOCATOR USAGE IS SQL TYPE IS CLOB-LOCATOR.
     01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.
     01 MY-BLOB USAGE IS SQL TYPE IS BLOB(1M).
     01 MY-BLOB-LOCATOR USAGE IS SQL TYPE IS BLOB-LOCATOR.
     01 MY-BLOB-FILE USAGE IS SQL TYPE IS BLOB-FILE.
     01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(1M).
     01 MY-DBCLOB-LOCATOR USAGE IS SQL TYPE IS DBCLOB-LOCATOR.
     01 MY-DBCLOB-FILE USAGE IS SQL TYPE IS DBCLOB-FILE.
     01 MY-PICTURE PIC G(16000) USAGE IS DISPLAY-1.
     01 dt         PIC X(10).
```

```
    01 tm         PIC X(8).
    01 tmstmp     PIC X(26).
    01 wage-ind   PIC S9(4) COMP-5.
  *
EXEC SQL END DECLARE SECTION END-EXEC.
```

다음은 지원되는 COBOL 데이터 유형에 대한 추가 규칙입니다.

- PIC S9 및 COMP-3/COMP-5는 표시되는 경우 필수입니다.
- VARCHAR, LONG VARCHAR, VARGRAPHIC, LONG VARGRAPHIC 및 모든 LOB 변수 유형을 제외한 모든 컬럼 유형에 대해 01 대신 77 레벨 번호를 사용할 수 있습니다.
- DECIMAL(p,s) 컬럼 유형에 대해 호스트 변수를 선언할 때 다음 규칙을 사용하십시오. 다음 샘플을 참조하십시오.

  ```
  01 identifier PIC S9(m)V9(n) COMP-3
  ```

  – V를 사용하여 소수점을 표시합니다.
  – *n* 및 *m*의 값은 1보다 크거나 같아야 합니다.
  – *n* + *m*의 값은 31을 초과할 수 없습니다.
  – *s*의 값은 *n*의 값과 같아야 합니다.
  – *p*의 값은 *n* + *m*의 값과 같아야 합니다.
  – 반복 인수 (*n*)과 (*m*)은 선택적입니다. 다음 예는 모두 유효합니다.

  ```
  01 identifier PIC S9(3)V COMP-3
  01 identifier PIC SV9(3) COMP-3
  01 identifier PIC S9V COMP-3
  01 identifier PIC SV9 COMP-3
  ```

  – PACKED-DECIMAL can be used instead of COMP-3.
- COBOL 사전 처리 컴파일러에서 배열은 지원하지 **않습니다**.

COBOL에서 FOR BIT DATA

특정 데이터베이스 컬럼은 2진 데이터용으로 선언할 수 있습니다. 일반적으로 문자를 포함한 이런 컬럼은 2진 정보를 유지하는 데 사용됩니다. CHAR(*n*), VARCHAR, LONG VARCHAR 및 BLOB 데이터 유형은 2진 데이터를 포함할 수 있는 COBOL 호스트 변수 유형입니다. 2진 데이터용 속성을 가진 컬럼에 대해 작업할 때 이런 데이터 유형을 사용하십시오.

COBOL에서 SQLSTATE 및 SQLCODE 변수

SQL92E 값에 LANGLEVEL 사전 처리 컴파일 옵션을 사용할 경우 다음 두 선언이 호스트 변수로 포함될 수 있습니다.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SQLSTATE PICTURE X(5).
01 SQLCODE  PICTURE S9(9) USAGE COMP.
.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
```

이들 중 어느 것도 지정하지 않을 경우, 사전 처리 컴파일 단계 중 SQLCODE 선언이 취해집니다. '01'은 '77'이 될 수도 있고 'PICTURE'는 'PIC'가 될 수도 있습니다. 이 옵션을 사용할 때 INCLUDE SQLCA문을 지정해서는 안 됩니다. 이 점에 주의하십시오.

다중 소스 파일로 구성된 응용프로그램의 경우 위와 같은 소스 파일에 SQLCODE 및 SQLSTATE 선언이 포함될 수 있습니다.

COBOL에 대한 일본어 또는 대만어 EUC 및 UCS-2 고려사항

UCS-2 데이터베이스에 연결되었거나 eucJp 또는 eucTW 코드 세트 아래에서 수행 중인 응용프로그램에서 전송된 모든 그래픽 데이터에는 UCS-2 코드 페이지 식별자 태그가 지정됩니다. 사용자의 응용프로그램에서 그래픽 문자열을 데이터베이스 서버로 전송하기 전에 먼저 UCS-2로 변환해야 합니다. 마찬가지로 임의의 응용프로그램이 UCS-2 데이터베이스에서 검색한 그래픽 데이터나 EUC eucJP 또는 eucTW 코드 페이지 아래에서 수행 중인 임의의 응용프로그램이 임의의 데이터베이스에서 검색한 그래픽 데이터는 UCS-2를 사용하여 엔코딩됩니다. 이때 사용자에게 UCS-2 데이터가 제공되지 않는 한 응용프로그램이 내부적으로 UCS-2에서 응용프로그램 코드 페이지로 변환해야 합니다.

SQLDA로 데이터를 복사하기 전에와 SQLDA에서 데이터를 복사한 후 이 변환을 실행해야 하므로 응용프로그램에서 UCS-2로의 또는 그 반대의 변환을 담당합니다. DB2 Universal Database에서는 사용자의 응용프로그램에 액세스할 수 있는 변환 루틴을 제공하지 않습니다. 대신 사용자가 운영 체제에서 사용할 수 있는

시스템 호출을 사용해야 합니다. UCS-2 데이터베이스의 경우 VARCHAR 및 VARGRAPHIC 스칼라 함수 사용을 고려할 수도 있습니다.

이런 함수에 대해서는 *SQL 참조서*에서 자세한 내용을 참조하십시오. 일반 EUC 응용프로그램 개발 지침에 대해서는 581 페이지의 『일본어 및 대만어 EUC 및 UCS-2 코드 세트 변환』에서 자세한 내용을 참조하십시오.

오브젝트 지향 COBOL

오브젝트 지향 COBOL을 사용할 경우 다음을 따라야 합니다.

- SQL문은 컴파일 단위의 첫번째 프로그램이나 클래스에만 나타날 수 있습니다. 사전 처리 컴파일러에서는 첫번째 작업 저장영역 절로 임시 작업 데이터를 삽입합니다.
- 오브젝트 지향 COBOL 프로그램에서, SQL문을 포함한 모든 클래스에는 비어 있는 경우에도 클래스 레벨 작업 저장영역 절이 있어야 합니다. 이 절은 사전 처리 컴파일러에서 생성하는 데이터 정의를 저장하는 데 사용합니다.

제24장 FORTRAN 프로그래밍

FORTRAN에 대한 프로그래밍 고려사항 787
FORTRAN에서 언어 제한사항 787
FORTRAN에서의 참조에 의한 호출 . . 788
FORTRAN에서 디버깅 및 주석 행 . . . 788
FORTRAN에 대한 프로그래밍 고려사항 788
FORTRAN에 대한 입력과 출력 파일 . . . 788
FORTRAN에 대한 포함 파일 789
FORTRAN에 파일 포함 792
FORTRAN에서의 SQL문 포함 793
FORTRAN에서 호스트 변수 794
FORTRAN에서의 호스트 변수 명명 . . 795
호스트 변수 선언 795
FORTRAN에서 표시기 변수 798
FORTRAN에서 LOB 선언 798
FORTRAN에서 LOB 위치 지정자 선언 799
FORTRAN에서 파일 참조 선언 800
FORTRAN에서 지원되는 SQL 데이터 유형 800
FORTAN에서 SQLSTATE 및 SQLCODE 변수 803
FORTRAN에서의 다중 바이트 문자 세트에 대한 고려사항 803
FORTRAN에 대한 일본어 또는 대만어 EUC 및 UCS-2 고려사항 803

FORTRAN에 대한 프로그래밍 고려사항

특수 호스트 언어 프로그래밍 고려사항은 다음 페이지에서 다룹니다. 언어 제한사항, 호스트 언어 고유의 include 파일, Embedded SQL문, 호스트 변수 및 호스트 변수에 대해 지원되는 데이터 유형에 대한 정보가 포함됩니다.

주: 미래에 DB2 버전 5에서 안정화된 FORTRAN 지원이 계획되어 있지만 FORTRAN 지원에 대한 추가 처리는 계획되어 있지 않습니다. 예를 들어, FORTRAN 사전 처리 컴파일러에서는 18 바이트 이상의 테이블 이름과 같은 SQL 오브젝트 식별자를 조절할 수 없습니다. 19 - 128 바이트의 테이블 이름과 같이 버전 5 이후에 DB2에 도입된 기능을 사용하려면 기타 FORTRAN으로 응용프로그램을 작성합니다.

FORTRAN에서 언어 제한사항

다음 절에서는 FORTRAN 언어 제한사항에 대해 설명합니다.

© Copyright IBM Corp. 1993, 2000

FORTRAN에서의 참조에 의한 호출

일부 API 매개변수는 호출 변수의 값보다 주소를 요구합니다. 데이터베이스 관리 프로그램에서는 GET ADDRESS, DEREFERENCE ADDRESS 및 COPY MEMORY API를 제공하는데 API는 이런 매개변수를 제공하는 사용자의 기능을 단순화합니다. 이런 API에 대한 완전한 설명은 *Administrative API Reference*의 내용을 참조하십시오.

FORTRAN에서 디버깅 및 주석 행

일부 FORTRAN 컴파일러에서는 컬럼 1의 'D' 또는 'd'가 포함된 행을 조건 행으로 처리합니다. 디버깅을 위해 또는 주석 행으로 처리하기 위해 이런 행을 컴파일할 수 있습니다. 사전 처리 컴파일러에서는 컬럼 1의 'D' 또는 'd'가 포함된 행을 항상 주석으로 처리합니다.

FORTRAN에 대한 프로그래밍 고려사항

다음 항목은 사전 처리 컴파일 프로세스에 영향을 줍니다.

- 사전 처리 컴파일러는 연속 행 컬럼 1-5에서 숫자, 공백 및 탭 문자만을 허용합니다.
- `.sqf` 소스 파일에서는 알파벳-숫자 코드(hollerith) 상수가 지원되지 않습니다.

사용자에게 영향을 줄 수 있는 다른 사전 처리 컴파일 고려사항에 대해서는 *응용 프로그램 빌드 안내서*에서 자세한 내용을 참조하십시오.

FORTRAN에 대한 입력과 출력 파일

기본적으로 입력 파일의 확장자는 `.sqf`이지만 TARGET 사전 처리 컴파일 옵션을 사용할 경우 입력 파일의 확장자를 선호하는 확장자로 지정할 수 있습니다.

기본적으로 출력 파일의 확장자는 UNIX 플랫폼의 경우 `.f`이고 OS/2 및 Windows 기반 플랫폼의 경우 `.for`이지만 OUTPUT 사전 처리 컴파일 옵션을 사용하여 출력 수정 소스 파일의 새로운 이름과 경로를 지정할 수 있습니다.

FORTRAN에 대한 포함 파일

FORTRAN용 호스트 언어 고유의 include 파일의 확장자는 UNIX의 경우 .f이고 OS/2의 경우 .for입니다. 응용프로그램에서 다음 FORTRAN include 파일을 사용할 수 있습니다.

SQL (sql.f)
이 파일에는 바인더, 사전 처리 컴파일러에 대한 언어 특정 프로토타입과 오류 메시지 API가 있습니다. 또한 시스템 상수를 정의합니다.

SQLAPREP (sqlaprep.f)
이 파일에는 사용자의 사전 처리 컴파일러를 작성하기 위해 요구되는 정의가 있습니다.

SQLCA (sqlca_cn.f, sqlca_cs.f)
이 파일은 SQLCA 구조를 정의합니다. SQLCA에는 SQL문과 API 호출의 실행에 관한 오류 정보와 함께 응용프로그램을 제공할 데이터베이스 관리 프로그램에 의해 사용되는 변수가 있습니다.

FORTRAN 응용프로그램용으로 두 SQLCA 파일이 제공됩니다. 기본 파일인 sqlca_cs.f에서는 IBM SQL 호환 형식으로 SQLCA 구조를 정의합니다. SQLCA NONE 옵션으로 사전 처리 컴파일된 sqlca_cn.f 파일에서는 더 나은 성능을 위해 SQLCA 구조를 정의합니다.

SQLCA_92 (sqlca_92.f)
이 파일에는 SQL 통신 영역(SQLCA) 구조의 FIPS SQL92 Entry Level 순응 버전이 있습니다. IPS SQL92 Entry Level 표준을 따르는 DB2 응용프로그램을 작성할 때 이 파일은 sqlca_cn.f 또는 sqlca_cs.f 파일 대신 포함되어야 합니다. LANGLEVEL 사전 처리 컴파일러 옵션을 SQL92E로 설정하면 DB2 사전 처리 컴파일러에서 sqlca_92.f 파일을 자동으로 포함합니다.

SQLCODES (sqlcodes.f)
이 파일은 SQLCA 구조의 SQLCODE에 대한 상수를 정의합니다.

SQLDA (sqldact.f)

이 파일은 SQLDA 구조를 정의합니다. SQLDA는 응용프로그램과 데이터베이스 관리 프로그램 사이에서 데이터를 전달하기 위해 사용됩니다. FORTRAN 프로그램에서 SQLDA를 코딩하는 방법에 대해서는 167 페이지의『SQLDA 구조 할당』에서 자세한 내용을 참조하십시오.

SQLEAU (sqleau.f)

이 파일에는 DB2 보안 감사 API에 요구되는 상수 및 구조 정의가 있습니다. 이러한 API를 사용하면 프로그램에서 이 파일을 포함시킬 필요가 있습니다. 또한 이 파일에는 감사 추적 레코드의 필드에 대한 상수 및 키워드 값 정의가 포함됩니다. 외부 또는 벤더 감사 추적 추출 프로그램에서 이런 정의를 사용할 수 있습니다.

SQLENV (sqlenv.f)

이 파일은 데이터베이스 환경 API용 언어 특정 호출과 그러한 인터페이스에 대한 구조, 상수 및 리턴 코드를 정의합니다.

SQLE819A (sqle819a.f)

데이터베이스의 코드 페이지가 819(ISO Latin-1)일 경우, 이 순서는 호스트 CCSID 500(EBCDIC 국제어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLE819B (sqle819b.f)

데이터베이스의 코드 페이지가 819(ISO Latin-1)일 경우, 이 순서는 호스트 CCSID 037(EBCDIC 미국 영어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLE850A (sqle850a.f)

데이터베이스의 코드 페이지가 850(ASCII Latin-1)일 경우, 이 순서는 호스트 CCSID 500(EBCDIC 국제어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLE850B (sqle850b.f)

데이터베이스의 코드 페이지가 850(ASCII Latin-1)일 경우, 이 순서는 호스트 CCSID 037(EBCDIC 미국 영어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLE932A (sqle932a.f)

데이터베이스의 코드 페이지가 932(ASCII Japanese)일 경우, 이 순서는 호스트 CCSID 5035(EBCDIC 일본어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLE932B (sqle932b.f)

데이터베이스의 코드 페이지가 932(ASCII Japanese)일 경우, 이 순서는 호스트 CCSID 5026(EBCDIC 일본어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQL1252A (sql1252a.f)

데이터베이스의 코드 페이지가 1252(Windows Latin-1)일 경우, 이 순서는 호스트 CCSID 500(EBCDIC 국제어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQL1252B (sql1252b.f)

데이터베이스의 코드 페이지가 1252(Windows Latin-1)일 경우, 이 순서는 호스트 CCSID 037(EBCDIC 미국 영어) 2진 집합에 따라 2진 데이터용이 아닌 문자열을 정렬합니다. 이 파일은 CREATE DATABASE API에 의해 사용됩니다.

SQLMON (sqlmon.f)

이 파일은 데이터베이스 시스템 모니터 API용 언어 특정 호출과 그러한 인터페이스에 대한 구조, 상수 및 리턴 코드를 정의합니다.

SQLSTATE (sqlstate.f)

이 파일은 SQLCA 구조의 SQLSTATE 필드에 대한 상수를 정의합니다.

SQLUTIL (sqlutil.f)

이 파일은 유틸리티 API용 언어 특정 호출과 그러한 인터페이스에 요구되는 구조, 상수 및 코드를 정의합니다.

FORTRAN에 파일 포함

파일을 포함하는 방법은 EXEC SQL INCLUDE문 사용과 FORTRAN INCLUDE 문 사용의 두 가지 방법이 있습니다. 사전 처리 컴파일러에서는 FORTRAN INCLUDE문을 무시하고, EXEC SQL INCLUDE문으로 포함한 파일만을 처리합니다.

INCLUDE 파일을 찾기 위해 DB2 FORTRAN 사전 처리 컴파일러에서는 먼저 현재 디렉토리를 검색한 다음 DB2INCLUDE 환경 변수로 지정된 디렉토리를 검색합니다. 다음 예를 고려하십시오.

- `EXEC SQL INCLUDE payroll`

  위와 같이 INCLUDE문에서 지정한 파일을 인용 부호로 닫지 않은 경우 사전 처리 컴파일러는 자신이 찾은 각 디렉토리에서 `payroll.sqf` 다음에 `payroll.f`(OS/2에서는 `payroll.for`) 순으로 검색합니다.

- `EXEC SQL INCLUDE 'pay/payroll.f'`

  위와 같이 파일 이름을 인용 부호로 닫을 경우 확장자는 이름에 추가되지 않습니다. OS/2의 경우 파일은 `'pay\payroll.for'`로 지정됩니다.

  인용 부호 안의 파일 이름에 절대 경로가 없을 경우 INCLUDE 파일 이름에 지정된 경로에 따라 DB2INCLUDE의 내용을 사용하여 파일을 검색합니다. 예를 들어, AIX용 DB2를 사용하여, DB2INCLUDE를 '`/disk2:myfiles/fortran`'로 설정하면 사전 처리 컴파일러에서 '`./pay/payroll.f`', '`/disk2/pay/payroll.f`', '`./myfiles/cobol/pay/payroll.f`' 순으로 검색합니다. 실제로 파일이 있는 경로가 사전 처리 컴파일러 메시지에 표시됩니다. OS/2의 경우 위의 예에서 정방향 슬래시를 역슬래시(\)로 대체하고 `'f'` 확장자를 `'for'`로 대체하십시오.

주: DB2 명령행 처리기에서 DB2INCLUDE의 설정을 캐쉬화합니다. CLP 명령을 발행한 후 DB2INCLUDE의 설정을 변경하려면 TERMINATE 명령을 입력한 다음 데이터베이스에 다시 연결하고 평소와 마찬가지로 사전 처리 컴파일하십시오.

FORTRAN에서의 SQL문 포함

Embedded SQL문은 다음 세 구성요소로 구성됩니다.

구성요소	올바른 FORTRAN 구문
키워드	EXEC SQL
명령문 문자열	분리문자로 공백을 포함한 유효한 SQL문
명령문 종료 표시자	소스 행의 끝

소스 행의 끝은 명령문 종료기 역할을 합니다. 행이 연속될 경우 명령문 종료기는 마지막 연속 행의 끝입니다.

예를 들면 다음과 같습니다.

```
EXEC SQL SELECT COL INTO :hostvar FROM TABLE
```

다음 규칙이 Embedded SQL문에 적용됩니다.

- 컬럼 7 - 72 사이에서만 SQL문을 코딩하십시오.
- 전체 행 FORTRAN 주석 또는 SQL 주석을 사용하지만 SQL문에서 FORTRAN 행의 끝(eol) 주석 '!' 문자를 사용하지 마십시오. 호스트 변수 선언을 포함하여, 이 주석 문자는 어느 곳에나 사용할 수 있습니다.
- FORTRAN문에서 분리문자로 공백을 요구하지 않더라도 Embedded SQL문을 코딩할 때 분리문자로 공백을 사용하십시오.
- 각 FORTRAN 소스 행에 대해 한 SQL문만을 사용하십시오. 일반 FORTRAN 연속 규칙은 소스 행을 하나 이상 요구하는 명령문에 대해 적용됩니다. 행 사이의 EXEC SQL 키워드 쌍을 분리하지 마십시오.
- Embedded SQL문의 일부인 모든 행에서 SQL 주석이 허용됩니다. 동적으로 실행되는 명령문에서는 이런 주석이 허용되지 않습니다. SQL 주석의 형식은 이중 대시(**--**) 다음에 0개 이상의 문자로 구성되며 행의 끝으로 종료됩니다.

- FORTRAN 주석은 Embedded SQL문 안의 *거의* 모든 위치에서 허용됩니다. 예외는 다음과 같습니다.
  - EXEC와 SQL 사이에서는 주석이 허용되지 않습니다.
  - 동적으로 실행되는 명령문에서는 주석이 허용되지 않습니다.
  - Embedded SQL문에서는 행의 끝에서 !의 확장자를 사용한 FORTRAN 주석 코딩을 지원하지 않습니다.
- SQL문에서 실제 상수를 지정할 때 지수 표기를 사용하십시오. 데이터베이스 관리 프로그램에서는 SQL문에서 소수점이 있는 숫자 문자열을 실질 상수가 아닌 소수 상수로 해석합니다.
- 명령문 번호는 첫번째 실행 가능 FORTRAN문 앞에 나오는 SQL문에서 유효합니다. SQL문에 그와 연관된 명령문 번호가 없을 경우 사전 처리 컴파일러에서 SQL문을 직접 처리하는, 레이블이 붙은 CONTINUE문을 생성합니다.
- SQL문에서 호스트 변수를 참조할 때 선언된 것과 동일한 호스트 변수를 사용하십시오.
- 행의 끝(eol)과 TAB 문자와 같은 공백 문자 대체는 다음과 같이 발생합니다.
  - SQL문 안이지만 외부 인용 부호 외부에 나타날 경우 행의 끝과 TAB은 단일 공백으로 대체됩니다.
  - 인용 부호 안에서 나타날 때 FORTRAN 프로그램에서 문자열이 제대로 연속되지 않을 경우 문자열이 행의 끝 문자가 사라집니다. TAB은 수정되지 않습니다.

  행의 끝과 TAB에 사용되는 실제 문자는 플랫폼에 따라 달라진다는 점에 주의하십시오. 예를 들어, OS/2에서는 행의 끝에 대해 캐리지 리턴이나 라인 피드를 사용하지만 UNIX 기반 시스템에서는 라인 피드만 사용합니다.

FORTRAN에서 호스트 변수

호스트 변수는 SQL문에서 참조되는 FORTRAN 언어 변수입니다. 이런 변수를 사용하여 응용프로그램에서 데이터베이스 관리 프로그램으로 입력 데이터를 전달하고 그로부터 출력 데이터를 수신할 수 있습니다. 응용프로그램을 사전 처리 컴파일하면 컴파일러에서 다른 FORTRAN 변수와 마찬가지로 호스트 변수를 사용합니다. 호스트 변수를 명명하고, 선언하고, 사용할 때 다음 제안을 활용하십시오.

FORTRAN에서의 호스트 변수 명명

SQL 사전 처리 컴파일러에서 선언된 이름으로 호스트 변수를 식별합니다. 다음 제한이 적용됩니다.

- 최대 255자까지 변수 이름을 지정하십시오.
- 호스트 변수 이름을 시스템에 사용할 수 있도록 예약된 SQL, sql, DB2 또는 db2 외의 접두사를 사용하십시오.

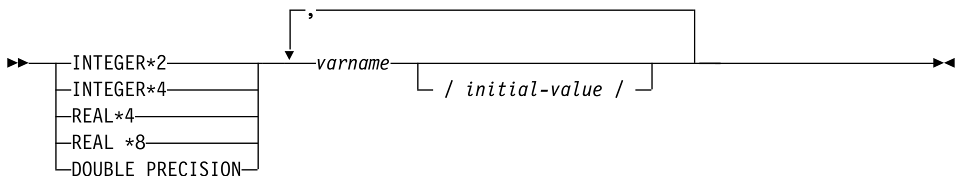
호스트 변수 선언

SQL 선언 절은 호스트 변수 선언을 식별하는 데 사용해야 합니다. 이 절은 후속 SQL문에서 참조할 수 있는 호스트 변수를 후속 사전 처리 컴파일러에 알려 줍니다.

FORTRAN 사전 처리 컴파일러만이 유효한 FORTRAN 선언의 부속 집합을 유효한 호스트 변수 선언으로 인식합니다. 이런 선언에서 숫자나 문자 변수를 정의합니다. 숫자 호스트 변수는 숫자 SQL 입력 또는 출력 값의 입력 또는 출력 변수로 사용할 수 있습니다. 문자 호스트 변수는 문자, 날짜, 시간 또는 시간소인 SQL 입력 또는 출력 값의 입력 또는 출력 변수로 사용할 수 있습니다. 프로그래머가 출력 변수가 수신한 값을 포함할만큼 충분히 긴 지 확인해야 합니다. FORTRAN에서 숫자 호스트 변수의 구문은 숫자 호스트 변수의 구문을 나타냅니다.

구조화 유형에 대해 호스트 변수를 선언하는 것에 대한 정보는 381 페이지의 『구조화 유형 호스트 변수 선언』을 참조하십시오.

FORTRAN에서 숫자 호스트 변수의 구문

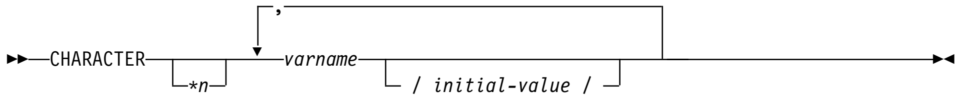


숫자 호스트 변수 고려사항:

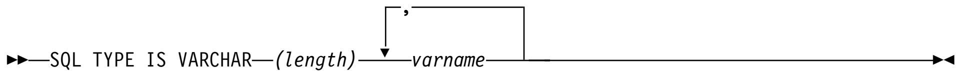
1. REAL*8 및 DOUBLE PRECISION은 동일합니다.
2. REAL*8 상수에 대한 지수 표시기로 D 대신 E를 사용하십시오.

FORTRAN에서 문자 호스트 변수의 구문: 고정 길이는 문자 호스트 변수의 구문을 나타냅니다.

FORTRAN에서 문자 호스트 변수의 구문: 고정 길이



변수 길이



문자 호스트 변수 고려사항:

1. *n의 최대 값은 254입니다.
2. 길이가 1과 32 672 사이일 경우 호스트 변수의 유형은 VARCHAR(SQLTYPE 448)입니다.
3. 길이가 32 673과 32 700 사이일 경우 호스트 변수 유형은 LONG VARCHAR(SQLTYPE 456)입니다.
4. 선언 내에서 VARCHAR 및 LONG VARCHAR 호스트 변수의 초기화는 허용되지 않습니다.

VARCHAR 예:

선언:

```
sql type is varchar(1000) my_varchar
```

그 결과 다음 구조가 생성됩니다.

```
 character    my_varchar(1000+2)
 integer*2    my_varchar_length
 character    my_varchar_data(1000)
 equivalence( my_varchar(1),
+             my_varchar_length )
 equivalence( my_varchar(3),
+             my_varchar_data )
```

예를 들어, 응용프로그램에서 my_varchar_length와 my_varchar_data를 조작하여 호스트 변수의 내용을 설정하거나 검사할 수 있습니다. VARCHAR를 전체로 참조하기 위해 기본 이름(이 경우, my_varchar)이 SQL문에서 사용됩니다.

LONG VARCHAR 예:

선언:

```
sql type is varchar(10000) my_lvarchar
```

그 결과 다음 구조가 생성됩니다.

```
 character    my_lvarchar(10000+2)
 integer*2    my_lvarchar_length
 character    my_lvarchar_data(10000)
 equivalence( my_lvarchar(1),
+             my_lvarchar_length )
 equivalence( my_lvarchar(3),
+             my_lvarchar_data )
```

예를 들어, 응용프로그램에서 my_lvarchar_length와 my_lvarchar_data를 조작하여 호스트 변수의 내용을 검사하거나 설정할 수 있습니다. VARCHAR를 전체로 참조하기 위해 기본 이름(이 경우, my_lvarchar)이 SQL문에서 사용됩니다.

주: 아래의 예와 같은 CONNECT문에서 FORTRAN 문자열 호스트 변수 dbname과 userid 뒤에 공백이 포함되지만 처리 전에 제거됩니다.

```
EXEC SQL CONNECT TO :dbname USER :userid USING :passwd
```

그러나 암호에서 공백이 중요할 수 있으므로 VARCHAR과 같은 암호에 호스트 변수를 선언하고 실제 암호 길이를 반영하도록 길이 필드를 설정해야 합니다.

```
     EXEC SQL BEGIN DECLARE SECTION
      character*8 dbname, userid
      sql type is varchar(18) passwd
 EXEC SQL END DECLARE SECTION
    character*18 passwd_string
    equivalence(passwd_data,passwd_string)
    dbname = 'sample'
    userid = 'userid'
    passwd_length= 8
    passwd_string = 'password'
    EXEC SQL CONNECT TO :dbname USER :userid USING :passwd
```

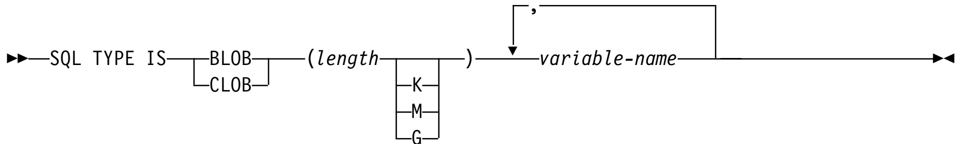
FORTRAN에서 표시기 변수

표시기 변수는 INTEGER*2 데이터 유형으로 선언해야 합니다.

FORTRAN에서 LOB 선언

FORTRAN에서 대형 오브젝트(LOB) 호스트 변수의 구문은 FORTRAN에서 대형 오브젝트(LOB) 호스트 변수를 선언하는 구문을 나타냅니다.

FORTRAN에서 대형 오브젝트(LOB) 호스트 변수의 구문



LOB 호스트 변수 고려사항:

1. GRAPHIC 유형은 FORTRAN에서 지원하지 않습니다.
2. SQL TYPE IS, BLOB, CLOB, K, M, G는 대문자, 소문자 또는 대소문자 혼합으로 나타날 수 있습니다.
3. BLOB and CLOB의 경우 1 <= lob-length <= 2 147 483 647
4. LOB 선언에서 LOB의 초기화는 허용되지 않습니다.
5. 사전 처리 컴파일러 생성 코드에서 호스트 변수 이름 앞에는 'length'와 'data'가 지정됩니다.

BLOB 예:

선언:

```
sql type is blob(2m) my_blob
```

그 결과 다음 구조가 생성됩니다.

```
 character    my_blob(2097152+4)
 integer*4    my_blob_length
 character    my_blob_data(2097152)
 equivalence( my_blob(1),
+             my_blob_length )
 equivalence( my_blob(5),
+             my_blob_data )
```

CLOB 예:

선언:

```
sql type is clob(125m) my_clob
```

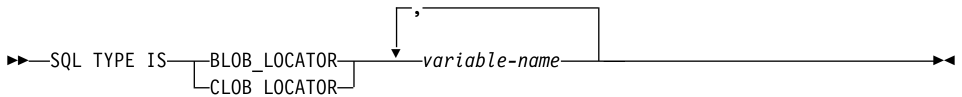
그 결과 다음 구조가 생성됩니다.

```
 character    my_clob(131072000+4)
 integer*4    my_clob_length
 character    my_clob_data(131072000)
 equivalence( my_clob(1),
+             my_clob_length )
 equivalence( my_clob(5),
+             my_clob_data )
```

FORTRAN에서 LOB 위치 지정자 선언

FORTRAN에서 대형 오브젝트(LOB) 위치 지정자 호스트 변수의 구문은 FORTRAN에서 대형 오브젝트(LOB) 위치 지정자 호스트 변수를 선언하는 구문을 나타냅니다.

FORTRAN에서 대형 오브젝트(LOB) 위치 지정자 호스트 변수의 구문



LOB 위치 지정자 호스트 변수 고려사항:

1. GRAPHIC 유형은 FORTRAN에서 지원하지 않습니다.
2. SQL TYPE IS, BLOB_LOCATOR, CLOB_LOCATOR은 대문자, 소문자 또는 대소문자 혼합으로 나타날 수 있습니다.
3. 위치 지정자의 초기화는 허용되지 않습니다.

CLOB 위치 지정자 예(BLOB 위치 지정자와 비슷):

선언:

```
SQL TYPE IS CLOB_LOCATOR my_locator
```

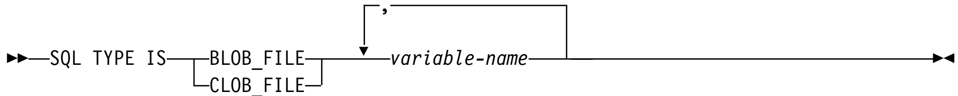
그 결과 다음 선언이 생성됩니다.

```
 integer*4 my_locator
```

FORTRAN에서 파일 참조 선언

FORTRAN에서 파일 참조 호스트 변수의 구문은 FORTRAN에서 파일 참조 호스트 변수를 선언하는 구문을 나타냅니다.

FORTRAN에서 파일 참조 호스트 변수의 구문



파일 참조 호스트 변수 고려사항:

1. Graphic 유형은 FORTRAN에서 지원하지 않습니다.
2. SQL TYPE IS, BLOB_FILE, CLOB_FILE은 대문자, 소문자 또는 대소문자 혼합으로 나타날 수 있습니다.

BLOB 파일 참조 변수의 예 (CLOB 파일 참조 변수와 비슷):

```
SQL TYPE IS BLOB_FILE my_file
```

그 결과 다음 선언이 생성됩니다.

```
 character     my_file(267)
 integer*4     my_file_name_length
 integer*4     my_file_data_length
 integer*4     my_file_file_options
 character*255 my_file_name
 equivalence(  my_file(1),
+              my_file_name_length )
 equivalence(  my_file(5),
+              my_file_data_length )
 equivalence(  my_file(9),
+              my_file_file_options )
 equivalence(  my_file(13),
+              my_file_name )
```

FORTRAN에서 지원되는 SQL 데이터 유형

사전 정의된 특정 FORTRAN 데이터 유형은 데이터베이스 관리 프로그램 컬럼 유형에 해당합니다. 이런 FORTRAN 데이터 유형은 호스트 변수로 선언할 수 있습니다.

표34는 각 컬럼 유형에 해당하는 FORTRAN 데이터 유형을 나타냅니다. 사전 처리 컴파일러에서 호스트 변수 선언을 찾으면 해당 SQL 유형 값을 판별합니다. 데이터베이스 관리 프로그램에서 이 값을 사용하여 응용프로그램과 자체에서 교환된 데이터를 변환합니다.

주: 모든 DB2 호스트 언어에서 DATALINK 데이터 유형에 대한 호스트 변수 지원은 없습니다.

표 34. FORTRAN 선언에 맵핑되는 SQL 데이터 유형

SQL 컬럼 유형[1]	FORTRAN 데이터 유형	SQL 컬럼 유형 설명
SMALLINT (500 또는 501)	INTEGER*2	부호있는 16비트 정수
INTEGER (496 또는 497)	INTEGER*4	부호있는 32비트 정수
REAL[2] (480 또는 481)	REAL*4	단정밀 부동 소수점
DOUBLE[3] (480 또는 481)	REAL*8	배정밀 부동 소수점
DECIMAL(*p*,*s*) (484 또는 485)	정확한 해당 데이터 유형 없음, REAL*8 사용	팩된 10진수
CHAR(*n*) (452 또는 453)	CHARACTER**n*	*n* 길이의 고정 길이 문자열. 여기서 *n*의 범위는 1 - 254.
VARCHAR(*n*) (448 또는 449)	SQL TYPE IS VARCHAR(*n*) 여기서 *n*의 범위는 1 - 32 672	가변 길이 문자열
LONG VARCHAR (456 또는 457)	SQL TYPE IS VARCHAR(*n*) 여기서 *n*의 범위는 32 673 - 32 700	긴 가변 길이 문자열
CLOB(*n*) (408 또는 409)	SQL TYPE IS CLOB (n) 여기서 *n*의 범위는 1 - 2 147 483 647	대형 오브젝트(LOB) 가변 길이 문자열
CLOB 위치 지정자 변수[4] (964 또는 965)	SQL TYPE IS CLOB_LOCATOR	서버의 CLOB 엔터티를 식별합니다.
CLOB 파일 참조 변수[4] (920 또는 921)	SQL TYPE IS CLOB_FILE	CLOB 데이터를 포함한 파일의 설명자
BLOB(*n*) (404 또는 405)	SQL TYPE IS BLOB(n) 여기서 *n*의 범위는 1 - 2 147 483 647	대형 오브젝트(LOB) 가변 길이 2진 문자열
BLOB 위치 지정자 변수[4] (960 또는 961)	SQL TYPE IS BLOB_LOCATOR	서버의 BLOB 엔터티를 식별합니다.
BLOB 파일 참조 변수[4] (916 또는 917)	SQL TYPE IS BLOB_FILE	BLOB 데이터를 포함한 파일의 설명자
DATE (384 또는 385)	CHARACTER*10	10바이트 문자열

표 34. FORTRAN 선언에 맵핑되는 SQL 데이터 유형 (계속)

SQL 컬럼 유형[1]	FORTRAN 데이터 유형	SQL 컬럼 유형 설명
TIME (388 또는 389)	CHARACTER*8	8바이트 문자열
TIMESTAMP (392 또는 393)	CHARACTER*26	26바이트 문자열

주:

1. **SQL 컬럼 유형** 아래서 첫번째 숫자는 표시기 변수가 제공되지 않은 경우를 나타내며 두 번째 숫자는 표시기 변수가 제공된 경우를 나타냅니다. 표시기 변수는 널(NULL) 값을 나타내거나 절단된 문자열의 길이를 유지하는 데 필요합니다. 이들은 해당 데이터 유형에 대해 SQLDA의 SQLTYPE에 나타나는 값입니다.
2. FLOAT(*n*) 여기서 $0 < n < 25$는 REAL의 동의어입니다. SQLDA에서 REAL과 DOUBLE 사이의 차이점은 길이 값(4 또는 8)입니다.
3. 다음 SQL 유형은 DOUBLE의 동의어입니다.
   - FLOAT
   - FLOAT(*n*) 여기서 $24 < n < 54$
   - DOUBLE PRECISION
4. 이는 컬럼 유형이 아닌 호스트 변수 유형입니다.

다음은 지원되는 각 데이터 유형에 대해 선언된 호스트 변수를 포함한 샘플 SQL 선언 절입니다.

```
    EXEC SQL BEGIN DECLARE SECTION
      INTEGER*2     AGE  /26/
      INTEGER*4     DEPT
      REAL*4        BONUS
      REAL*8        SALARY
      CHARACTER     MI
      CHARACTER*112 ADDRESS
      SQL TYPE IS VARCHAR (512) DESCRIPTION
      SQL TYPE IS VARCHAR (32000) COMMENTS
      SQL TYPE IS CLOB (1M) CHAPTER
      SQL TYPE IS CLOB_LOCATOR CHAPLOC
      SQL TYPE IS CLOB_FILE  CHAPFL
      SQL TYPE IS BLOB (1M) VIDEO
      SQL TYPE IS BLOB_LOCATOR VIDLOC
      SQL TYPE IS BLOB_FILE VIDFL
      CHARACTER*10  DATE
      CHARACTER*8   TIME
      CHARACTER*26  TIMESTAMP
      INTEGER*2     WAGE_IND
  EXEC SQL END DECLARE SECTION
```

다음은 지원되는 FORTRAN 데이터 유형에 대한 추가 규칙입니다.

- VARCHAR, LONG VARCHAR, OR CLOB 호스트 변수를 사용하여 254 자 이상의 동적 SQL문을 정의할 수 있습니다.

FORTAN에서 SQLSTATE 및 SQLCODE 변수

SQL92E 값에 LANGLEVEL 사전 처리 컴파일 옵션을 사용할 경우 다음 두 선언이 호스트 변수로 포함될 수 있습니다.

```
EXEC SQL BEGIN DECLARE SECTION;
  CHARACTER*5 SQLSTATE
  INTEGER     SQLCOD
  .
  .
  .
EXEC SQL END DECLARE SECTION
```

이들 중 어느 것도 지정하지 않을 경우 사전 처리 컴파일 단계 중 SQLCOD 선언이 취해집니다. 'SQLSTATE'의 변수 이름이 'SQLSTA'가 될 수도 있습니다. 이 옵션을 사용할 때 INCLUDE SQLCA문을 지정해선 안됩니다. 이 점에 주의하십시오.

복수 소스 파일이 있는 응용프로그램의 경우 위와 같은 각 소스 파일에 SQLCOD 및 SQLSTATE 선언이 포함될 수 있습니다.

FORTRAN에서의 다중 바이트 문자 세트에 대한 고려사항

FORTRAN에서는 그래픽(복수 바이트) 호스트 변수 데이터 유형을 지원하지 않습니다. 대소문자가 혼합된 호스트 변수만을 문자 데이터 유형을 통해 지원합니다. 그래픽 데이터가 있는 사용자 SQLDA를 작성할 수 있습니다.

FORTRAN에 대한 일본어 또는 대만어 EUC 및 UCS-2 고려사항

UCS-2 데이터베이스에 연결되었거나 eucJp 또는 eucTW 코드 세트 아래에서 수행 중인 응용프로그램에서 전송된 모든 그래픽 데이터에는 UCS-2 코드 페이지 식별자 태그가 지정됩니다. 사용자의 응용프로그램에서 그래픽 문자열을 데이터베이스 서버로 전송하기 전에 먼저 UCS-2로 변환해야 합니다. 마찬가지로 임의의 응용프로그램이 UCS-2 데이터베이스에서 검색한 그래픽 데이터나 EUC eucJP 또

는 eucTW 코드 페이지 아래에서 수행 중인 임의의 응용프로그램이 임의의 데이터베이스에서 검색한 그래픽 데이터는 UCS-2를 사용하여 엔코딩됩니다. 이때 사용자에게 UCS-2 데이터가 제공되지 않는 한 응용프로그램이 내부적으로 UCS-2에서 응용프로그램 코드 페이지로 변환해야 합니다.

SQLDA로 데이터를 복사하기 전에와 SQLDA에서 데이터를 복사한 후 이 변환을 실행해야 하므로 응용프로그램에서 UCS-2로의 또는 그 반대의 변환을 담당합니다. DB2 Universal Database에서는 사용자의 응용프로그램에 액세스할 수 있는 변환 루틴을 제공하지 않습니다. 대신 사용자가 운영 체제에서 사용할 수 있는 시스템 호출을 사용해야 합니다. UCS-2 데이터베이스의 경우 VARCHAR 및 VARGRAPHIC 스칼라 함수 사용을 고려할 수도 있습니다.

이런 함수에 대해서는 *SQL 참조서*에서 자세한 내용을 참조하십시오.

일반 EUC 응용프로그램 개발 지침에 대해서는 581 페이지의『일본어 및 대만어 EUC 및 UCS-2 코드 세트 변환』에서 자세한 내용을 참조하십시오.

제25장 REXX 프로그래밍

REXX에 대한 프로그래밍 고려사항. 805
REXX에 대한 언어 제한사항. 806
REXX에서 SQLEXEC, SQLDBS 및 SQLDB2 등록 806
REXX에서 Embedding SQL문 808
REXX에서 호스트 변수 810
REXX에서 호스트 변수 명명. 810
REXX에서 호스트 변수 조회. 810
REXX의 표시기 변수 811
사전 정의된 REXX 변수 811
REXX의 LOB 호스트 변수 813
REXX에서 LOB 위치 지정자 선언. . . 814
REXX에서 LOB 파일 참조 선언 . . . 815
REXX에서 LOB 호스트 변수 지움. . . 816
REXX에서 지원되는 SQL 데이터 유형 . . 817
REXX에서 커서 사용 819
REXX에 대한 실행 요구사항. 819
REXX에 대한 바인드 파일 820
REXX용 API 구문 821
REXX저장 프로시듀어 823
REXX에서 저장 프로시듀어 호출 . . . 823
REXX용 클라이언트에 대한 고려사항 825
REXX용 서버에 대한 고려사항 . . . 825
SQLDA 10진 필드에서 정밀도 및 SCALE 값 검색 825
REXX에 대한 일본어 또는 대만어 EUC 고려사항 826

REXX에 대한 프로그래밍 고려사항

특수 호스트 언어 프로그래밍 고려사항은 다음 페이지에서 다룹니다. Embedded SQL문, 언어 제한사항 및 호스트 변수에 대해 지원되는 데이터 유형에 대한 정보가 포함됩니다.

주: 미래에 DB2 버전 5에서 안정화된 REXX 지원이 계획되어 있지만 REXX 지원에 대한 추가 처리는 계획되어 있지 않습니다. 예를 들어, REXX에서는 18 바이트 이상의 테이블 이름과 같은 SQL 오브젝트 식별자를 조절할 수 없습니다. 19 - 128 바이트의 테이블 이름과 같이 버전 5 이후에 DB2에 도입된 기능을 사용하려면 기타 REXX으로 응용프로그램을 작성합니다.

REXX가 해석된 언어이므로 사전 처리 컴파일러, 컴파일러 또는 링커를 사용하지 않습니다. 대신 세 개의 DB2 API를 사용하여 REXX로 DB2 응용프로그램을 작성합니다. 이런 API를 사용하여 DB2의 서로 다른 구성요소에 액세스하십시오.

SQLEXEC

SQL 언어를 지원합니다.

SQLDBS

DB2 API의 명령어와 같은 버전을 지원합니다.

SQLDB2

REXX 고유의 명령행 프로세서 인터페이스를 지원합니다. 이 인터페이스의 사용 방법에 대해서는 821 페이지의 『REXX용 API 구문』에서 자세한 내용을 참조하십시오.

REXX에 대한 언어 제한사항

SQLEXEC, SQLDBS 및 SQLDB2 루틴으로 전달되는 명령문 또는 명령의 토큰이 REXX 변수와 상응할 수 있습니다. 이 경우, REXX 인터프리터에서 SQLEXEC, SQLDBS 또는 SQLDB2를 호출하기 전에 변수의 값을 대체합니다.

이런 상황을 방지하려면 ' '나 " "와 같은 인용 부호로 명령문 문자열을 닫으십시오. 인용 부호를 사용하지 않을 경우 SQLEXEC, SQLDBS 또는 SQLDB2 루틴으로 전달하는 대신 REXX 인터프리터에서 충돌하는 변수 이름을 해결합니다.

REXX/SQL에서는 복합 SQL을 지원하지 않습니다.

OS/2 및 Windows 32 비트 운영 체제에서는 REXX/SQL 저장 프로시듀어를 지원하지만 AIX에서는 지원하지 않습니다.

REXX에서 SQLEXEC, SQLDBS 및 SQLDB2 등록

응용프로그램에서 DB2 API를 사용하거나 SQL문을 발행하기 전에 SQLDBS, SQLDB2 및 SQLEXEC 루틴을 등록해야 합니다. 그러면 REXX 인터프리터에 REXX/SQL 입력점이 통지됩니다. OS/2와 AIX 플랫폼에서 등록에 사용하는 방법은 약간 다릅니다. 다음 예는 각 루틴 등록의 올바른 구문을 나타냅니다.

OS/2 또는 Windows에서 샘플 등록

```
/* ------------ Register SQLDBS with REXX  --------------------------*/
If Rxfuncquery('SQLDBS')  <> 0 then
    rcy = Rxfuncadd('SQLDBS','DB2AR','SQLDBS')
 If rcy \= 0 then
   do
      say 'SQLDBS was not successfully added to the REXX environment'
     signal rxx_exit
```

```
   end

/* ------------ Register SQLDB2 with REXX  -------------------------*/
If Rxfuncquery('SQLDB2')  <> 0 then
    rcy = Rxfuncadd('SQLDB2','DB2AR','SQLDB2')
 If rcy \= 0 then
   do
      say 'SQLDB2 was not successfully added to the REXX environment'
     signal rxx_exit
   end

/* ----------------- Register SQLEXEC with REXX  ---------------------*/
If Rxfuncquery('SQLEXEC') <> 0 then
    rcy = Rxfuncadd('SQLEXEC','DB2AR','SQLEXEC')
 If rcy \= 0 then
   do
      say 'SQLEXEC was not successfully added to the REXX environment'
     signal rxx_exit
   end
```

AIX에서 샘플 등록

```
 /* ----- Register SQLDBS, SQLDB2 and SQLEXEC with REXX -----*/
 rcy = SysAddFuncPkg("db2rexx")
 If rcy \= 0 then
   do
     say 'db2rexx was not successfully added to the REXX environment'
     signal rxx_exit
   end
```

OS/2에서 RxFuncAdd 명령은 모든 세션에 대해 한번만 실행해야 합니다.

AIX의 경우 모든 REXX/SQL 응용프로그램에서 SysAddFuncPkg를 실행해야 합니다.

RXfuncadd 및 SysAddFuncPkg API에 대해서는 각기 OS/2 및 AIX용 REXX 설명서를 참조하십시오.

REXX에서 Embedding SQL문

SQLEXEC 루틴을 사용하면 모든 SQL문을 처리할 수 있습니다. SQLEXEC 루틴에 대한 문자열 인수는 다음 구성요소로 구성됩니다.

- SQL 키워드
- 사전 선언된 식별자
- 명령문 호스트 변수

SQLEXEC 루틴으로 유효한 SQL문을 전달하여 각 요청을 작성하십시오. 다음 구문을 사용하십시오.

```
CALL SQLEXEC 'statement'
```

SQL문이 한 행 이상으로 연속될 수 있습니다. 다음과 같이 명령문의 각 부분은 단일 인용 부호로 닫아야 하며 추가 명령문 텍스트는 쉼표로 분리해야 합니다.

```
CALL SQLEXEC 'SQL text',
             'additional text',
                  .
                  .
                  .
             'final text'
```

다음은 REXX에서의 SQL문 Embedding의 예입니다.

```
statement = "UPDATE STAFF SET JOB = 'Clerk' WHERE JOB = 'Mgr'"
CALL SQLEXEC 'EXECUTE IMMEDIATE :statement'
IF ( SQLCA.SQLCODE < 0) THEN
   SAY 'Update Error:  SQLCODE = ' SQLCA.SQLCODE
```

이 예에서 SQLCA 구조의 SQLCODE 필드를 점검하여 갱신이 성공했는지 여부를 판별합니다.

다음 규칙이 Embedded SQL문에 적용됩니다.

- 다음 SQL문을 SQLEXEC 루틴으로 직접 전달할 수 있습니다.

  CALL

  CLOSE

  COMMIT

  CONNECT

  CONNECT TO

CONNECT RESET
DECLARE
DESCRIBE
DISCONNECT
EXECUTE
EXECUTE IMMEDIATE
FETCH
FREE LOCATOR
OPEN
PREPARE
RELEASE
ROLLBACK
SET CONNECTION

EXECUTE IMMEDIATE 또는 PREPARE 및 EXECUTE문을 SQLEXEC 루틴과 결합해 사용하여 다른 SQL문을 동적으로 처리해야 합니다.

- REXX에서 CONNECT 및 SET CONNECTION의 호스트 변수를 사용할 수 없습니다.
- 커서 이름과 명령문 이름은 다음과 같이 사전 정의되어 있습니다.

c1에서 c100

커서 이름으로, WITH HOLD 옵션을 설정하지 않고 선언한 커서의 경우 *c1 - c50* 범위를 가지며 WITH HOLD 옵션을 사용하여 선언한 커서의 경우 *c51 - c100* 범위를 가집니다.

커서 이름 식별자는 DECLARE, OPEN, FETCH 및 CLOSE문에 사용됩니다. 이는 SQL 요청에 사용된 커서를 식별합니다.

s1에서 s100

명령문 이름으로 범위는 *s1 - s100*입니다.

명령문 이름 식별자는 DECLARE, DESCRIBE, PREPARE 및 EXECUTE문과 함께 사용됩니다.

커서와 명령문 이름에 사전 선언한 식별자를 사용해야 합니다. 다른 이름은 허용되지 않습니다.

- 커서 선언시 DECLARE문에서 커서 이름과 명령문 이름은 상응해야 합니다. 예를 들어, 커서 이름으로 *c1*을 사용할 경우 명령문 이름에 *s1*을 사용해야 합니다.
- SQL문 안에서 주석을 사용하지 마십시오.

REXX에서 호스트 변수

호스트 변수는 SQL문에서 참조되는 REXX 언어 변수입니다. 이런 변수를 사용하여 응용프로그램에서 DB2로 입력 데이터를 전달하고 DB2에서 출력 데이터를 수신할 수 있습니다. LOB 위치 지정자 및 LOB 파일 참조 변수를 제외하면 REXX 응용프로그램에서 호스트 변수를 선언할 필요가 없습니다. 호스트 변수 데이터 유형 및 크기는 변수가 참조되는 런타임에 결정됩니다. 호스트 변수를 명명하고 사용할 때 다음 규칙을 적용하십시오.

REXX에서 호스트 변수 명명

적절히 명명된 REXX 변수를 호스트 변수로 사용할 수 있습니다. 변수 이름의 길이는 64자까지 가능합니다. 마침표로 이름을 종료하지 마십시오. 호스트 변수 이름은 알파벳 문자, 숫자 및 @, _, !, ., ?, $ 등의 문자로 구성될 수 있습니다.

REXX에서 호스트 변수 조회

REXX 인터프리터가 프로시듀어에서 인용 부호가 없는 모든 문자열을 검사합니다. 문자열이 현재 REXX 변수 풀의 변수를 표시하면 REXX에서 문자열을 현재 값으로 대체합니다. 다음은 REXX에서 호스트 변수를 참조하는 방법의 한 예입니다.

```
CALL SQLEXEC 'FETCH C1 INTO :cm'
SAY 'Commission = ' cm
```

문자열이 숫자 데이터 유형으로 변환되지 않게 하려면 다음 예에서와 같이 문자열을 단일 인용 부호로 닫으십시오.

```
VAR = '100'
```

REXX에서는 *VAR* 변수를 3 바이트 문자열 100으로 설정합니다. 단일 인용 부호를 문자열의 일부로 포함시키려는 경우 다음 예를 따르십시오.

```
VAR = "'100'"
```

CHARACTER 필드에 숫자 데이터를 삽입하면 REXX 인터프리터에서 숫자 데이터를 정수로 처리하므로 숫자 문자열을 명시적으로 연결하고 단일 인용 부호로 닫아야 합니다.

REXX의 표시기 변수

REXX의 표시기 변수 데이터 유형은 소수점이 없는 숫자입니다. 다음은 INDICATOR 키워드를 사용하는 REXX의 표시기 변수의 예입니다.

```
CALL SQLEXEC 'FETCH C1 INTO :cm INDICATOR :cmind'
IF ( cmind < 0 )
   SAY 'Commission is NULL'
```

위의 예에서 cmind의 값이 음수인지 검사합니다. 음수가 아닐 경우 응용프로그램에서 cm의 리턴된 값을 사용할 수 있습니다. 음수일 경우 가져온 값은 널(NULL)이므로 cm을 사용해선 안됩니다. 이 경우 데이터베이스 관리 프로그램에서는 호스트 변수의 값을 변경하지 않습니다.

사전 정의된 REXX 변수

SQLEXEC, SQLDBS 및 SQLDB2는 사전 정의된 REXX 변수를 특정 조작의 결과로 설정합니다. 이런 변수는 다음과 같습니다.

RESULT

각 조작은 이 리턴 코드를 설정합니다. 가능한 값은 다음과 같습니다.

n 여기서 *n*은 형식화된 메시지의 바이트 수를 지시하는 양의 값입니다. GET ERROR MESSAGE API에서만 이 값을 리턴합니다.

0 API가 실행되었습니다. REXX 변수 SQLCA는 API의 완전한 상태를 포함합니다. SQLCA.SQLCODE가 0이 아닐 경우 SQLMSG에는 그 값과 연관된 텍스트 메시지가 있습니다.

–1 API를 완료하는 데 사용 가능한 메모리가 충분하지 않습니다. 요청된 메시지가 리턴되지 않았습니다.

–2 SQLCA.SQLCODE가 0으로 설정됩니다. 메시지가 리턴되지 않았습니다.

–3 SQLCA.SQLCODE에 유효하지 않은 SQLCODE가 있었습니다. 메시지가 리턴되지 않았습니다.

–6 SQLCA REXX 변수를 빌드할 수 없습니다. 즉, 사용 가능한 메모리가 충분하지 않거나 어떤 이유로 REXX 변수 풀을 사용할 수 없습니다.

–7 SQLMSG REXX 변수를 빌드할 수 없습니다. 즉, 사용 가능한 메모리가 충분하지 않거나 어떤 이유로 REXX 변수 풀을 사용할 수 없습니다.

–8 REXX 변수 풀에서 SQLCA.SQLCODE REXX 변수를 가져올 수 없습니다.

–9 가져오는 중 SQLCA.SQLCODE REXX 변수가 절단되었습니다. 이 변수의 최대 길이는 5 바이트입니다.

–10 SQLCA.SQLCODE REXX 변수를 ASCII에서 유효한 long 정수로 변환할 수 없습니다.

–11 REXX 변수 풀에서 SQLCA.SQLERRML REXX 변수를 가져올 수 없습니다.

–12 가져오는 중 SQLCA.SQLERRML REXX 변수가 절단되었습니다. 이 변수의 최대 길이는 2 바이트입니다.

–13 SQLCA.SQLERRML REXX 변수를 ASCII에서 유효한 short 정수로 변환할 수 없습니다.

–14 REXX 변수 풀에서 SQLCA.SQLERRMC REXX 변수를 가져올 수 없습니다.

–15 가져오는 중 SQLCA.SQLERRMC REXX 변수가 절단되었습니다. 이 변수의 최대 길이는 70 바이트입니다.

–16 오류 텍스트용으로 지정된 REXX 변수를 설정할 수 없습니다.

–17 REXX 변수 풀에서 SQLCA.SQLSTATE REXX 변수를 가져올 수 없습니다.

–18 가져오는 중 SQLCA.SQLSTATE REXX 변수가 절단되었습니다. 이 변수의 최대 길이는 2 바이트입니다.

주: GET ERROR MESSAGE API에서만 –8 - –18 사이의 값을 리턴합니다.

SQLMSG

SQLCA.SQLCODE가 0이 아닐 경우 이 변수에는 오류 코드와 연관된 텍스트 메시지가 있습니다.

SQLISL

분리 레벨. 가능한 값은 다음과 같습니다.

RR 반복 읽기(RR)

RS 읽기 안정성(RS)

CS 커서 안정성(CS). 기본값입니다.

UR 미확약 읽기(UR)

NC 확약하지 않습니다.(일부 호스트 또는 AS/400 서버에서만 NC를 지원합니다.)

SQLCA

SQL문 이후에 갱신된 SQLCA 구조가 처리되고 DB2 API가 호출됩니다. 이 구조의 항목은 *Administrative API Reference*에 설명되어 있습니다.

SQLRODA

CALL문을 사용하여 호출된 저장 프로시듀어의 입력/출력 SQLDA 구조. 또한 데이터베이스 응용프로그램 원격 인터페이스(DARI) API를 사용하여 호출된 저장 프로시듀어의 출력 SQLDA 구조이기도 합니다. 이 구조의 항목은 *Administrative API Reference*에 설명되어 있습니다.

SQLRIDA

데이터베이스 응용프로그램 원격 인터페이스(DARI) API를 사용하여 호출된 저장 프로시듀어의 입력 SQLDA 구조입니다. 이 구조의 항목은 *Administrative API Reference*에 설명되어 있습니다.

SQLRDAT

데이터베이스 응용프로그램 원격 인터페이스(DARI) API를 사용하여 호출된 서버 프로시듀어의 SQLCHAR 구조입니다. 이 구조의 항목은 *Administrative API Reference*에 설명되어 있습니다.

REXX의 LOB 호스트 변수

LOB 컬럼을 REXX 호스트 변수로 가져오면 이는 단순(즉, uncounted) 문자열로 저장됩니다. 이는 CHAR, VARCHAR, GRAPHIC, LONG 등과 같은 모든 문자 기반 SQL 유형의 경우와 같은 방식으로 조절됩니다. 입력시 호스트 변수 내용의 크기가 32K보다 크거나 아래 설정된 다른 기준을 만족시킬 경우 해당 LOB 유형으로 할당됩니다.

REXX SQL에서 LOB 유형은 아래와 같이 호스트 변수의 문자열 내용에 따라 결정됩니다.

호스트 변수 문자열 내용	결과 LOB 유형
:hv1='32K보다 긴 일반 인용 부호로 닫힌 문자열 ...'	CLOB
:hv2="'인용 부호 분리가 Embedded된 문자열 ", "32K보다 긴...'"	CLOB
:hv3="문자열 분리가 Embedded된 G'DBCS 문자열 ", "인용 부호, G로 시작, 32K보다 긴 문자열...'"	DBCLOB
:hv4="문자열 분리가 Embedded된 BIN'문자열 ", "인용 부호, BIN으로 시작,임의 길이...'"	BLOB

REXX에서 LOB 위치 지정자 선언

REXX에서 LOB 위치 지정자 호스트 변수의 구문은 REXX에서 LOB 위치 지정자 호스트 변수를 선언하는 구문을 나타냅니다.

REXX에서 LOB 위치 지정자 호스트 변수의 구문

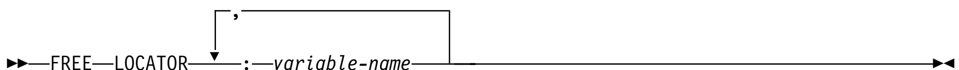
응용프로그램에서 LOB 위치 지정자 호스트 변수를 선언해야 합니다. REXX/SQL에서 이런 선언을 발견하면 선언된 호스트 변수를 나머지 프로그램의 위치 지정자로 처리합니다. 위치 지정자 값은 REXX 변수에 내부 형식으로 저장됩니다.

예를 들면 다음과 같습니다.

```
CALL SQLEXEC 'DECLARE :hv1, :hv2 LANGUAGE TYPE CLOB LOCATOR'
```

엔진에서 리턴된 LOB 위치 지정자가 표시하는 데이터는 다음과 같은 형식의 FREE LOCATOR문을 사용하여 REXX/SQL에서 해제할 수 있습니다.

FREE LOCATOR문에 대한 구문



예를 들면 다음과 같습니다.

```
CALL SQLEXEC 'FREE LOCATOR :hv1, :hv2'
```

REXX에서 LOB 파일 참조 선언

응용프로그램에서 LOB 파일 참조 호스트 변수를 선언해야 합니다. REXX/SQL에서 이런 선언을 발견하면 선언된 호스트 변수를 나머지 프로그램의 LOB 파일 참조로 처리합니다.

REXX에서 LOB 파일 참조 변수의 구문은 REXX에서 LOB 파일 참조 호스트 변수를 선언하는 구문을 나타냅니다.

REXX 파일 참조 선언

예를 들면 다음과 같습니다.

```
CALL SQLEXEC 'DECLARE :hv3, :hv4 LANGUAGE TYPE CLOB FILE'
```

REXX의 파일 참조 변수에는 세 필드가 있습니다. 위의 예의 경우 다음과 같습니다.

hv3.FILE_OPTIONS.
파일 사용 방법을 지시하도록 응용프로그램에서 설정합니다.

hv3.DATA_LENGTH.
파일 크기를 지시하도록 DB2에서 설정합니다.

hv3.NAME.
LOB 파일의 이름을 지시하도록 응용프로그램에서 설정합니다.

FILE_OPTIONS에 대해 응용프로그램에서는 다음과 같은 키워드를 설정합니다.

키워드(정수 값)
의미

READ (2)
: 입력에 사용할 파일. 이 파일은 열고, 읽고, 닫을 수 있는 일반 파일입니다. 파일의 데이터 길이(바이트 단위)는 파일을 열 때 계산됩니다(응용프로그램 요청자 코드에 따라).

CREATE (8)
: 출력시, 새로운 파일을 작성합니다. 파일이 이미 있으면 오류가 발생합니다. 파일의 길이(바이트 단위)는 파일 참조 변수 구조의 DATA_LENGTH 필드에 리턴됩니다.

OVERWRITE (16)
: 출력시 기존 파일이 있는 경우 기존 파일은 겹쳐쓰여지고, 그렇지 않은 경우 새로운 파일이 작성됩니다. 파일의 길이(바이트 단위)는 파일 참조 변수 구조의 DATA_LENGTH 필드에 리턴됩니다.

APPEND (32)
: 출력은 파일이 있는 경우 그 파일에 추가되고, 그렇지 않은 경우 새로운 파일이 작성됩니다. 전체 파일 길이가 아닌 파일에 추가되는 데이터 길이(바이트 단위)는 파일 참조 변수 구조의 DATA_LENGTH 필드에 리턴됩니다.

주: 파일 참조 호스트 변수는 REXX의 혼합 변수이므로 NAME, NAME_LENGTH 및 FILE_OPTIONS 필드를 선언할 뿐 아니라 해당 필드의 값도 설정해야 합니다.

REXX에서 LOB 호스트 변수 지움

OS/2의 경우 REXX SQL LOB 위치 지정자와 파일 참조 호스트 변수 선언을 그대로 두면 응용프로그램 종료 후에 영향을 미치므로 명시적으로 지워야 할 경우가 있습니다. 응용프로그램이 수행 중인 세션을 닫기 전까지 응용프로그램 프로세스가 나가지 않기 때문입니다. REXX SQL LOB 선언을 지우지 않으면 LOB 응용프로그램 실행 후 동일 세션에서 수행 중인 다른 응용프로그램과 상호작용할 수 있습니다.

선언을 지우는 구문은 다음과 같습니다.

```
CALL SQLEXEC "CLEAR SQL VARIABLE DECLARATIONS"
```

LOB 응용프로그램의 끝에서 이 명령문을 코딩해야 합니다. 이전 응용프로그램에서 남겼을 수 있는 선언을 지우는 예방책으로, REXX SQL 응용프로그램 시작시와 같은 어느 위치에서든 명령문을 코딩할 수 있다는 점에 주의하십시오.

REXX에서 지원되는 SQL 데이터 유형

사전 정의된 특정 REXX 데이터 유형은 DB2 컬럼 유형에 해당합니다. 표35는 SQLEXEC 및 SQLDBS에서 REXX 변수의 내용을 DB2 데이터 유형으로 변환하기 위해 그 변수를 해석하는 방법을 나타냅니다.

주: 모든 DB2 호스트 언어에서 DATALINK 데이터 유형에 대한 호스트 변수 지원은 없습니다.

표 35. REXX 선언에 맵핑되는 SQL 컬럼 유형

SQL 컬럼 유형[1]	REXX 데이터 유형	SQL 컬럼 유형 설명
SMALLINT (500 또는 501)	소수점이 없는 숫자로 범위는 -32 768 - 32 767입니다.	부호있는 16비트 정수
INTEGER (496 또는 497)	소수점이 없는 숫자로 범위는 -2 147 483 648 - 2 147 483 647입니다.	부호있는 32비트 정수
REAL[2] (480 또는 481)	과학 기호법의 숫자로 범위는 -3.40282346 x 10^{38} - 3.40282346 x 10^{38}입니다.	단정밀 부동 소수점
DOUBLE[3] (480 또는 481)	과학 기호법의 숫자로 범위는 -1.79769313 x 10^{308} - 1.79769313 x 10^{308}입니다.	배정밀 부동 소수점
DECIMAL(*p*,*s*) (484 또는 485)	소수점이 있는 숫자	팩된 10진수
CHAR(*n*) (452 또는 453)	앞과 뒤에 인용 부호(')가 있는 문자열로, 두 인용 부호를 제거한 후 길이는 *n* 과학 기호법의 E 또는 앞 공백과 뒤 공백 이외의 숫자 문자를 포함하지 않는 *n* 길이의 문자열	*n* 길이의 고정 길이 문자열. 여기서 *n*의 범위는 1 - 254입니다.
VARCHAR(*n*) (448 또는 449)	CHAR에 해당(*n*)	*n* 길이의 가변 길이 문자열. 여기서 *n*의 범위는 1 - 4000입니다.
LONG VARCHAR (456 또는 457)	CHAR에 해당(*n*)	*n* 길이의 가변 길이 문자열. 여기서 *n*의 범위는 1 - 32 700입니다.
CLOB(*n*) (408 또는 409)	CHAR에 해당(*n*)	*n* 길이의 대형 오브젝트(LOB) 가변 길이 문자열. 여기서 *n*의 범위는 1 - 2 147 483 647입니다.
CLOB 위치 지정자 변수[4] (964 또는 965)	DECLARE *:var_name* LANGUAGE TYPE CLOB LOCATOR	서버의 CLOB 엔터티를 식별합니다.
CLOB 파일 참조 변수[4] (920 또는 921)	DECLARE *:var_name* LANGUAGE TYPE CLOB FILE	CLOB 데이터를 포함한 파일의 설명자
BLOB(*n*) (404 또는 405)	앞과 뒤에 어스트로피가 있는 문자열로 문자열 앞에는 BIN 지정. BIN과 두 개의 어스트로피를 제거한 후 문자열 길이는 *n*입니다.	*n* 길이의 대형 오브젝트(LOB) 가변 길이 2진 문자열. 여기서 *n*의 범위는 1 - 2 147 483 647입니다.

표 35. REXX 선언에 맵핑되는 SQL 컬럼 유형 (계속)

SQL 컬럼 유형[1]	REXX 데이터 유형	SQL 컬럼 유형 설명
BLOB 위치 지정자 변수[4] (960 또는 961)	DECLARE *:var_name* LANGUAGE TYPE BLOB LOCATOR	서버의 BLOB 엔터티를 식별합니다.
BLOB 파일 참조 변수[4] (916 또는 917)	DECLARE *:var_name* LANGUAGE TYPE BLOB FILE	BLOB 데이터를 포함한 파일의 설명자
DATE (384 또는 385)	CHAR에 해당(*10*)	10바이트 문자열
TIME (388 또는 389)	CHAR에 해당(*8*)	8바이트 문자열
TIMESTAMP (392 또는 393)	CHAR에 해당(*26*)	26바이트 문자열
주: 다음 데이터 유형은 DBCS 환경에서만 사용할 수 있습니다.		
GRAPHIC(*n*) (468 또는 469)	앞과 뒤에 어스트로피가 있는 문자열로 문자열 앞에는 G 또는 N 지정. 앞의 문자와 두 개의 어스트로피를 제거한 후 DBCS 문자는 *n*개입니다.	*n* 길이의 고정 길이 그래픽 문자열. 여기서 *n*의 범위는 1 - 127입니다.
VARGRAPHIC(*n*) (464 또는 465)	GRAPHIC에 해당(*n*)	*n* 길이의 가변 길이 그래픽 문자열. 여기서 *n*의 범위는 1 - 2000입니다.
LONG VARGRAPHIC (472 또는 473)	GRAPHIC에 해당(*n*)	*n* 길이의 긴 가변 길이 그래픽 문자열. 여기서 *n*의 범위는 1 - 16 350입니다.
DBCLOB(*n*) (412 또는 413)	GRAPHIC에 해당(*n*)	*n* 길이의 대형 오브젝트(LOB) 가변 길이 그래픽 문자열. 여기서 *n*의 범위는 1 - 1 073 741 823입니다.
DBCLOB 위치 지정자 변수[4] (968 또는 969)	DECLARE *:var_name* LANGUAGE TYPE DBCLOB LOCATOR	서버의 DBCLOB 엔터티를 식별합니다.
DBCLOB 파일 참조 변수[4] (924 또는 925)	DECLARE *:var_name* LANGUAGE TYPE DBCLOB FILE	DBCLOB 데이터를 포함한 파일의 설명자

주:

1. **컬럼 유형** 아래의 첫번째 숫자는 표시기 변수가 제공되지 않음을 지시하며 두 번째 숫자는 표시기 변수가 제공됨을 지시합니다. 표시기 변수는 널(NULL) 값을 나타내거나 절단된 문자열의 길이를 유지하는 데 필요합니다.
2. FLOAT(*n*) 여기서 $0 < n < 25$는 REAL의 동의어입니다. SQLDA에서 REAL과 DOUBLE 사이의 차이점은 길이 값(4 또는 8)입니다.
3. 다음 SQL 유형은 DOUBLE의 동의어입니다.
   - FLOAT
   - FLOAT(*n*) 여기서 $24 < n < 54$
   - DOUBLE PRECISION
4. 이는 컬럼 유형이 아닌 호스트 변수 유형입니다.

REXX에서 커서 사용

REXX에서 커서를 선언하면 커서는 조회와 연관됩니다. 조회는 PREPARE문에 할당되는 명령문 이름과 연관됩니다. 호스트 변수에 대한 모든 참조는 매개변수 표시문자로 표시됩니다. 다음 예에서는 동적 SELECT문과 연관된 DECLARE문을 나타냅니다.

```
prep_string = "SELECT TABNAME FROM SYSCAT.TABLES WHERE TABSCHEMA = ?"
CALL SQLEXEC 'PREPARE S1 FROM :prep_string';
CALL SQLEXEC 'DECLARE C1 CURSOR FOR S1';
CALL SQLEXEC 'OPEN C1 USING :schema_name';
```

REXX에 대한 실행 요구사항

REXX 응용프로그램은 사전 처리 컴파일되거나 컴파일되거나 링크되지 않습니다.

OS/2의 경우 응용프로그램 파일의 확장자는 .CMD여야 합니다. 파일 작성 후 운영 체제 명령 프롬프트에서 응용프로그램을 직접 수행할 수 있습니다.

Windows 32 비트 운영 체제의 경우 응용프로그램 파일은 임의의 이름을 가질 수 있습니다. 파일 작성 후 다음과 같이 REXX 인터프리터를 호출하여 운영 체제 명령 프롬프트에서 응용프로그램을 수행할 수 있습니다.

```
REXX file_name
```

AIX의 경우 응용프로그램 파일은 임의의 이름을 가질 수 있습니다. 다음과 같은 두 방법 중 하나를 사용하여 응용프로그램을 수행할 수 있습니다.

1. 쉘 명령 프롬프트에서 rexx name를 입력하십시오. 여기서 name은 REXX 프로그램의 이름입니다.
2. REXX 프로그램의 첫번째 행에는 "magic number"(#!)이 있으며 이 행에서 REXX/6000 인터프리터가 있는 디렉토리를 식별할 경우, 쉘 명령 프롬프트에서 그 이름을 입력하여 REXX 프로그램을 수행할 수 있습니다. 예를 들어, REXX/6000 인터프리터 파일이 /usr/bin 디렉토리에 있을 경우 REXX 프로그램의 첫번째 행에 다음을 포함시키십시오.

   ```
   #! /usr/bin/rexx
   ```

   그런 후 쉘 명령 프롬프트에서 다음 명령을 입력하여 프로그램 실행 파일을 작성하십시오.

```
chmod +x name
```

쉘 명령 프롬프트에서 이 파일 이름을 입력하여 REXX 프로그램을 수행하십시오.

주: AIX의 경우 REXX SQL 라이브러리 db2rexx가 있는 디렉토리를 포함하도록 LIBPATH 환경 변수를 설정해야 합니다. 예를 들면 다음과 같습니다.

```
export LIBPATH=/lib:/usr/lib:/usr/lpp/db2_07_01/lib
```

REXX에 대한 바인드 파일

REXX 응용프로그램을 지원하도록 다섯 개의 바인드 파일이 제공되었습니다. 이런 파일의 이름은 DB2UBIND.LST 파일에 있습니다. 각 바인드 파일은 다른 분리 레벨을 사용하여 사전 처리 컴파일되므로 데이터베이스에 다섯 개의 다른 패키지가 저장됩니다.

다섯 개의 바인드 파일은 다음과 같습니다.

DB2ARXCS.BND
: 커서 안정성(CS) 분리 레벨을 지원합니다.

DB2ARXRR.BND
: 반복 읽기(RR) 분리 레벨을 지원합니다.

DB2ARXUR.BND
: 미확약 읽기(UR) 분리 레벨을 지원합니다.

DB2ARXRS.BND
: 읽기 안정성(RS) 분리 레벨을 지원합니다.

DB2ARXNC.BND
: 확약되지 않은 분리 레벨을 지원합니다. 일부 호스트 또는 AS/400 데이터베이스 서버에 대해 작업할 때 분리 레벨이 사용됩니다. 다른 데이터베이스에서는 미확약 읽기(UR) 분리 레벨과 같이 수행합니다.

주: 몇몇 경우에 이런 파일을 데이터베이스에 명시적으로 바인드해야 할 수 있습니다.

SQLEXEC 루틴을 사용할 경우 커서 안정성으로 작성된 패키지가 기본으로 사용됩니다. 다른 분리 레벨 중 하나가 필요할 경우 데이터베이스에 연결하기 전에 SQLDBS CHANGE SQL ISOLATION LEVEL API로 분리 레벨을 변경할 수 있습니다. 그러면 SQLEXEC 루틴에 대한 후속 호출이 지정된 분리 레벨과 연관됩니다.

OS/2 REXX 응용프로그램에서는 세션에 있는 다른 REXX 응용프로그램에서 설정을 변경하지 않았다는 사실을 인식하지 않는 한 기본 분리 레벨이 적용된다고 가정할 수 없습니다. 데이터베이스에 연결하기 전에 REXX 응용프로그램에서 분리 레벨을 명시적으로 설정해야 합니다.

REXX용 API 구문

SQLDBS 루틴을 사용하여 다음 구문으로 DB2 API를 호출하십시오.

```
CALL SQLDBS 'command string'
```

DB2 API 작동 방식에 대해서는 *Administrative API Reference*의 DB2 API 장을 참조하십시오.

사용하려는 DB2 API를 SQLDB 루틴으로 호출할 수 없는 경우 즉, 해당 DB2 API가 *Administrative API Reference*에 나열되어 있지 않은 경우 REXX 응용프로그램에서 DB2 명령행 프로세서(CLP)를 호출하여 API를 호출할 수 있습니다. 그러나 DB2 CLP에서 출력을 표준 출력 장치나 지정된 파일로 지정하므로 REXX 응용프로그램에서 호출된 DB2 API의 출력에 직접 액세스하거나 호출된 API가 성공했는지 여부를 쉽게 판별할 수 없습니다. SQLDB2 API에서 각 호출 후에 복합 REXX 변수를 설정하여 각 호출의 성공 또는 실패시 REXX 응용프로그램에 직접 피드백을 제공하는 DB2 CLP의 인터페이스를 제공합니다.

SQLDB2 루틴을 사용하여 다음 구문으로 DB2 API를 호출할 수 있습니다.

```
CALL SQLDB2 'command string'
```

여기서 `'command string'`은 CLP에서 처리할 수 있는 문자열입니다. CLP에서 처리할 수 있는 문자열 구문에 대해서는 *Command Reference*에서 자세한 내용을 참조하십시오.

SQLDB2를 사용한 DB2 API 호출은 다음 경우를 제외한 CLP 직접 호출에 해당합니다.

- CLP 실행 파일 호출은, 다른 CLP 옵션과 매개변수를 동일하게 지정했을 때 SQLDB2 호출로 대체됩니다.
- REXX 복합 변수 SQLCA는 SQLDB2 호출 후 설정되지만 CLP 실행 파일 호출 후에는 설정되지 않습니다.
- SQLDB2를 호출할 때 CLP의 기본 표시 출력은 오프로 설정되지만 CLP 실행 파일을 호출할 때에는 표시가 온 출력으로 설정됩니다. SQLDB2로 +o 또는 -o- 옵션을 전달하여 CLP의 표시 출력을 온으로 설정할 수 있다는 점에 주의하십시오.

SQLDB2 호출 후 설정된 REXX 변수만이 SQLCA이므로 이 루틴을 사용하여, SQLCA 이외의 다른 데이터를 리턴하지 않고 현재 SQLDBS 인터페이스를 통해 구현되지 않은 DB2 API를 호출할 수 있습니다. 따라서 SQLDB2에서는 다음 DB2 API만 지원합니다.

Activate Database
Add Node
Bind for DB2 Version 1(1) (2)
Bind for DB2 Version 2 or 5(1)
Create Database at Node
Drop Database at Node
Drop Node Verify
Deactivate Database
Deregister
Load(3)
Load Query
Precompile Program(1)
Rebind Package(1)
Redistribute Nodegroup
Register
Start Database Manager
Stop Database Manager

SQLDB2에서 지원하는 DB2 API에 대한 주:

1. 이런 명령은 SQLDB2 인터페이스를 사용한 CONNECT문을 요구합니다. SQLDB2 인터페이스를 사용한 연결로는 SQLDB2 인터페이스에 액세스할 수 없으며 SQLEXEC 인터페이스를 사용한 연결로는 SQLDB2 인터페이스에 액세스할 수 없습니다.
2. OS/2에서 SQLDB2 인터페이스를 통해 지원됩니다.
3. Load API용 선택적 출력 매개변수 `pLoadInfoOut`는 REXX 응용프로그램으로 리턴되지 않습니다. Load API 및 그 매개변수에 대해서는 *Administrative API Reference*에서 자세한 내용을 참조하십시오.

주: SQLDB2 루틴의 목적이 위에 나열된 DB2 API 전용을 위한 것이지만 SQLDBS 루틴을 통해 지원되지 않는 다른 DB2 API에도 사용할 수 있습니다. 또는 REXX 응용프로그램에서 CLP을 통해 DB2 API에 액세스할 수 있습니다.

REXX저장 프로시듀어

REXX SQL 응용프로그램에서 SQL CALL문을 사용하여 데이터베이스 서버의 저장 프로시듀어를 호출할 수 있습니다. AIX 시스템의 REXX를 제외하면, 해당 서버에서 지원하는 모든 언어로 저장 프로시듀어를 작성할 수 있습니다. AIX 시스템에서 REXX로 클라이언트 응용프로그램을 작성할 수 있지만 다른 언어로 작성할 경우 응용프로그램은 AIX에서 REXX로 작성한 저장 프로시듀어를 호출할 수 없습니다.

REXX에서 저장 프로시듀어 호출

CALL문을 사용하여 클라이언트 응용프로그램에서 서버 저장 프로시듀어로 데이터를 전달하고 서버 저장 프로시듀어에서 데이터를 수신할 수 있습니다. 입력 및 출력 데이터의 인터페이스는 호스트 변수의 목록입니다. *SQL 참조서*에서 자세한 내용을 참조하십시오. REXX에서 일반적으로 그 내용에 기초하여 호스트 변수의 유형과 크기를 판별하므로 CALL로 전달되는 출력 전용 변수는 유형과 크기면에서 예상된 출력과 비슷한 *dummy* 데이터로 초기화해야 합니다.

CALL문의 USING DESCRIPTOR 구문을 사용하여 SQLDA REXX 변수를 통해 저장 프로시듀어에 데이터를 전달할 수도 있습니다. 표36은 SQLDA의 설정 방법을 나타냅니다. 테이블에서 ':value'는 응용프로그램에 필요한 값을 포함한 REXX 호스트 변수의 스템입니다. ' DESCRIPTOR의 경우 'n'은 SQLDA의 특정 *sqlvar* 구성요소를 지시하는 숫자 값입니다. 오른쪽의 숫자는 표36 뒤에 나오는 주를 의미합니다.

표 36. CALL문을 사용한 저장 프로시듀어용 클라이언트 측 REXX SQLDA

USING DESCRIPTOR	:value.SQLD	1	
	:value.n.SQLTYPE	1	
	:value.n.SQLLEN	1	
	:value.n.SQLDATA	1	2
	:value.n.SQLDIND	1	2

주:

1. 저장 프로시듀어를 호출하기 전에 클라이언트 응용프로그램에서 해당 데이터로 REXX 변수를 초기화해야 합니다.

   SQL CALL문을 실행하면 데이터베이스 관리 프로그램에서 저장영역을 할당하고 REXX 변수 풀에서 REXX 변수의 값을 검색합니다. CALL문에 사용된 SQLDA의 경우 데이터베이스 관리 프로그램에서 SQLTYPE 및 SQLLEN 값에 기초하여 SQLDATA 및 SQLIND 필드용 저장영역을 할당합니다.

   REXX 저장 프로시듀어 즉, 호출 중인 프로시듀어가 OS/2 REXX로 작성된 자신일 경우 DARI API 또는 CALL문의 유형에서 클라이언트가 전달한 데이터는 다음과 같은 사전 정의된 이름을 사용하여 데이터베이스 서버의 REXX 변수 풀에 들어갑니다.

   SQLRIDA

   REXX 입력 SQLDA 변수의 사전 정의된 이름

   SQLRODA

   REXX 출력 SQLDA 변수의 사전 정의된 이름

2. 저장 프로시듀어가 종료하면 데이터베이스 관리 프로그램에서 저장 프로시듀어의 변수 값을 검색합니다. 값은 클라이언트 응용프로그램으로 리턴되며 클라이언트의 REXX 변수 풀에 들어갑니다.

REXX용 클라이언트에 대한 고려사항

CALL문에서 호스트 변수를 사용할 경우 서버 프로시듀어에서 호스트 변수로 리턴되는 데이터와 호환 가능한 유형의 값으로 각 호스트 변수를 초기화하십시오. 해당 표시기가 음수일 경우에도 이 초기화를 수행해야 합니다.

설명자를 사용할 경우 SQLDATA를 초기화해야 하며 SQLDATA에는 서버 프로시듀어에서 리턴하는 데이터와 호환 가능한 유형의 데이터가 포함되어야 합니다. SQLIND 필드에 음수 값이 있는 경우에도 이 초기화를 수행해야 합니다.

REXX용 서버에 대한 고려사항

사전 정의된 출력 sqlda SQLRODA의 모든 SQLDATA 필드 및 SQLIND(널(NULL) 입력 가능한 경우)를 초기화했는지 확인하십시오. 예를 들어, SQLRODA.SQLD가 2일 경우 다음 필드에 어떤 데이터가 있어야 합니다(해당 표시기가 음수이고 데이터가 클라이언트로 다시 전달되지 않는 경우에도).

- SQLRODA.1.SQLDATA
- SQLRODA.2.SQLDATA

SQLDA 10진 필드에서 정밀도 및 SCALE 값 검색

데이터베이스 관리 프로그램에서 리턴하는 SQLDA 구조에서 십진수 필드의 정밀도 및 스케일 값을 검색하려면 REXX 프로그램에서 SQLDA 출력을 초기화할 때 `sqllen.scale` 및 `sqllen.precision` 값을 사용하십시오. 예를 들면 다음과 같습니다.

```
.
.
.
/* INITIALIZE ONE ELEMENT OF OUTPUT SQLDA */
io_sqlda.sqld = 1
io_sqlda.1.sqltype = 485        /* DECIMAL DATA TYPE */
io_sqlda.1.sqllen.scale = 2     /* DIGITS RIGHT OF DECIMAL POINT */
io_sqlda.1.sqllen.precision = 7 /* WIDTH OF DECIMAL */
io_sqlda.1.sqldata = 00000.00   /* HELPS DEFINE DATA FORMAT */
io_sqlda.1.sqlind = -1          /* NO INPUT DATA */
.
.
.
```

REXX에 대한 일본어 또는 대만어 EUC 고려사항

일본어나 대만어 EUC 환경에서는 REXX 응용프로그램이 지원되지 않습니다.

제7부 부록 및 끝머리

© Copyright IBM Corp. 1993, 2000

부록A. 지원되는 SQL문

표37:

- Linux, OS/2, UNIX 및 Windows 32-비트 운영 체제에 대해 DB2 Universal Database에서 지원되는 모든 SQL문을 나열합니다.
- 'X'로, 동적으로 실행할 수 있는지 여부를 지시합니다.
- 'X'로, 명령행 프로세서(CLP)에서 지원하는지 여부를 지시합니다.
- 'X' 또는 DB2 CLI 함수 이름으로, DB2 Call Level Interface(DB2 CLI)를 사용하여 명령문을 실행할 수 있는지 여부를 지시합니다.
- 명령문을 SQL 프로시듀어에서 실행할 수 있는지 나타냅니다('X' 사용).

표37을 빠른 참조 지원 도구로 사용할 수 있습니다. 그 구문을 포함하여 모든 명령문에 대한 완전한 설명은 *SQL 참조서*의 내용을 참조하십시오.

표 37. SQL문(DB2 Universal Database)

SQL문	동적[1]	명령행 처리기(CLP)	콜 레벨 인터페이스[3] (CLI)	SQL 프로시듀어
ALLOCATE CURSOR				X
명령문 지정				X
ASSOCIATE LOCATORS				X
ALTER { BUFFERPOOL, NICKNAME,[10] NODEGROUP, SERVER,[10] TABLE, TABLESPACE, USER MAPPING,[10] TYPE, VIEW }	X	X	X	
BEGIN DECLARE SECTION[2]				
CALL		X[9]	X[4]	X
CASE문				X
CLOSE		X	SQLCloseCursor(), SQLFreeStmt()	X
COMMENT ON	X	X	X	X
COMMIT	X	X	SQLEndTran, SQLTransact()	X
복합 SQL (Embedded)			X[4]	
복합문				X

© Copyright IBM Corp. 1993, 2000

표 37. SQL문(DB2 Universal Database) (계속)

SQL문	동적[1]	명령행 처리기(CLP)	콜 레벨 인터페이스[3] (CLI)	SQL 프로시듀어
CONNECT(유형 1)		X	SQLBrowseConnect(), SQLConnect(), SQLDriverConnect()	
CONNECT(유형 2)		X	SQLBrowseConnect(), SQLConnect(), SQLDriverConnect()	
CREATE { ALIAS, BUFFERPOOL, DISTINCT TYPE, EVENT MONITOR, FUNCTION, FUNCTION MAPPING,[10] INDEX, INDEX EXTENSION, METHOD, NICKNAME,[10] NODEGROUP, PROCEDURE, SCHEMA, TABLE, TABLESPACE, TRANSFORM TYPE MAPPING,[1] TRIGGER, USER MAPPING,[10] TYPE, VIEW, WRAPPER[10] }	X	X	X	X[11]
DECLARE CURSOR[2]		X	SQLAllocStmt()	X
DECLARE GLOBAL TEMPORARY TABLE	X	X	X	X
DELETE	X	X	X	X
DESCRIBE[8]		X	SQLColAttributes(), SQLDescribeCol(), SQLDescribParam()[6]	
DISCONNECT		X	SQLDisconnect()	
DROP	X	X	X	X[11]
END DECLARE SECTION[2]				
EXECUTE			SQLExecute()	X
EXECUTE IMMEDIATE			SQLExecDirect()	X
EXPLAIN	X	X	X	X
FETCH		X	SQLExtendedFetch()[7], SQLFetch(), SQLFetchScroll()[7]	X
FLUSH EVENT MONITOR	X	X	X	
FOR문				X
FREE LOCATOR			X[4]	X
GET DIAGNOSTICS				X

표 37. SQL문(DB2 Universal Database) (계속)

SQL문	동적[1]	명령행 처리기(CLP)	콜 레벨 인터페이스[3] (CLI)	SQL 프로시듀어
GOTO문				X
GRANT	X	X	X	X
IF문				X
INCLUDE[2]				
INSERT	X	X	X	X
ITERATE				X
LEAVE문				X
LOCK TABLE	X	X	X	X
LOOP문				X
OPEN		X	SQLExecute(), SQLExecDirect()	X
PREPARE			SQLPrepare()	X
REFRESH TABLE	X	X	X	
RELEASE		X		X
RELEASE SAVEPOINT	X	X	X	X
RENAME TABLE	X	X	X	
RENAME TABLESPACE	X	X	X	
REPEAT문				X
RESIGNAL문				X
RETURN문				X
REVOKE	X	X	X	
ROLLBACK	X	X	SQLEndTran(), SQLTransact()	X
SAVEPOINT	X	X	X	X
select-statement	X	X	X	X
SELECT INTO				X
SET CONNECTION		X	SQLSetConnection()	
SET CURRENT DEFAULT TRANSFORM GROUP	X	X	X	X
SET CURRENT DEGREE	X	X	X	X
SET CURRENT EXPLAIN MODE	X	X	X, SQLSetConnectAttr()	X
SET CURRENT EXPLAIN SNAPSHOT	X	X	X, SQLSetConnectAttr()	X
SET CURRENT PACKAGESET				

표 37. SQL문(DB2 Universal Database) (계속)

SQL문	**동적**[1]	**명령행 처리기(CLP)**	**콜 레벨 인터페이스**[3] **(CLI)**	**SQL 프로시듀어**
SET CURRENT QUERY OPTIMIZATION	X	X	X	X
SET CURRENT REFRESH AGE	X	X	X	X
SET EVENT MONITOR STATE	X	X	X	X
SET INTEGRITY	X	X	X	
SET PASSTHRU[10]	X	X	X	X
SET PATH	X	X	X	X
SET SCHEMA	X	X	X	X
SET SERVER OPTION[10]	X	X	X	X
SET transition-variable[5]	X	X	X	X
SIGNAL문				X
SIGNAL SQLSTATE[5]	X	X	X	
UPDATE	X	X	X	X
VALUES INTO				X
WHENEVER[2]				
WHILE문				X

표 37. SQL문(DB2 Universal Database) (계속)

SQL문	동적[1]	명령행 처리기(CLP)	콜 레벨 인터페이스[3] (CLI)	SQL 프로시듀어

주:

1. 이 목록의 모든 명령문으로 정적 SQL로 코딩할 수 있지만 X 표시된 명령문은 동적 SQL로 코딩하십시오.
2. 이 명령문을 실행할 수 없습니다.
3. X는 `SQLExecDirect()`나 `SQLPrepare()` 및 `SQLExecute()`를 사용하여 이 명령문을 실행할 수 있음을 나타냅니다. 해당 DB2 CLI 기능이 있을 경우 함수 이름이 나열됩니다.
4. 이 명령문이 동적이지는 않지만 `SQLExecDirect()`나, `SQLPrepare()` 및 `SQLExecute()`를 호출할 때 DB2 CLI를 사용하여 이 명령문을 지정할 수 있습니다.
5. CREATE TRIGGER문에서만 이를 사용할 수 있습니다.
6. SQL DESCRIBE문을 사용해서만 출력을 기술할 수 있지만 DB2 CLI를 사용하면 `SQLDescribeParam()` 함수로 입력도 기술할 수 있습니다.
7. SQL FETCH문을 사용하여 한 방향으로 한번에 한 행씩 가져올 수 있지만 DB2 CLI `SQLExtendedFetch()` 및 `SQLFetchScroll()` 함수를 사용해서는 배열로 가져올 수 있습니다. 더 나아가 어떤 방향으로든 결과 세트의 어떤 위치에서든 가져올 수 있습니다.
8. DESCRIBE SQL문의 구문은 CLP DESCRIBE 명령의 구문과 다릅니다. DESCRIBE SQL문에 대해서는 *SQL 참조서*에서 자세한 내용을 참조하십시오. DESCRIBE CLP 명령에 대해서는 *Command Reference*에서 자세한 내용을 참조하십시오.
9. 명령행 프로시듀어를 통해 CALL을 발행할 경우 다음 프로시듀어와 그 매개변수만 지원됩니다. (747 페이지의『JAR 파일 설치, 대체 및 제거』 참조).
10. 명령문은 연합 데이터베이스 서버에 대해서만 지원됩니다.
11. SQL 프로시듀어는 색인, 테이블 및 뷰에 대해서만 CREATE 및 DROP문을 실행할 수 있습니다.

부록B. 샘플 프로그램

이 절에는 DB2와 함께 제공되는 샘플 프로그램에 대한 정보가 있습니다. 모든 샘플 프로그램은 `sqllib` 디렉토리의 `samples` 서브디렉토리에 있습니다. 지원되는 각 언어에 대한 서브디렉토리가 있습니다.

이 책에서 사용한 샘플 프로그램은 지원되는 호스트 언어에서 Embedded SQL문 및 API 호출의 예를 나타냅니다. 샘플 프로그램은 짧고 단순하게 작성했습니다. 제작 응용프로그램에서는 리턴 코드 특히, API 호출 및 SQL문의 SQLCODE 또는 SQLSTATE를 점검합니다. 오류 조건, SQLCODE 및 SQLSTATE 조절에 대해서는 133 페이지의『진단 조절 및 SQLCA 구조』의 내용을 참조하십시오. 사용자 환경에서 이런 프로그램의 설치, 빌드 및 실행에 대해서는 *응용프로그램 빌드 안내서*에서 자세한 내용을 참조하십시오.

주:

1. 이 절에서는 DB2에서 지원하는 모든 플랫폼용 프로그래밍 언어의 샘플 프로그램에 대해 설명합니다. 모든 샘플 프로그램이 모든 플랫폼이나 지원되는 모든 프로그래밍 언어로 포트되는 것은 아닙니다.
2. DB2 샘플 프로그램은 그 어떠한 보증도 없이 "있는 그대로" 제공됩니다. IBM이 아닌 사용자가 전체 품질, 성능의 위험을 감수하고 결함을 정정합니다.

샘플 프로그램은 DB2 응용프로그램 개발(DB2 AD) 클라이언트와 함께 제공됩니다. 샘플 프로그램을 템플리트로 사용하여 사용자의 응용프로그램을 작성할 수 있습니다.

샘플 프로그램 파일 확장은 지원되는 각 언어 그리고 각 언어의 Embedded SQL 및 비embedded SQL 프로그램에서 다릅니다. 파일 확장자는 언어에서 프로그램 그룹에 대해서도 다를 수 있습니다. 이렇게 서로 다른 샘플 파일 확장은 다음과 같은 테이블로 분류됩니다.

언어별 샘플 파일 확장자

837 페이지의 표38.

© Copyright IBM Corp. 1993, 2000

프로그램 그룹별 샘플 파일 확장자
    838 페이지의 표39.

다음 테이블에서는 유형별 샘플 프로그램을 설명합니다.

Embedded SQL이 없는 DB2 API 샘플 프로그램
    840 페이지의 표40.

DB2 API Embedded SQL 샘플 프로그램
    844 페이지의 표41.

DB2 API가 없는 Embedded SQL 샘플 프로그램
    846 페이지의 표42.

사용자 정의 함수 샘플 프로그램
    847 페이지의 표43

DB2 CLI 샘플 프로그램
    848 페이지의 표44.

Java JDBC 샘플 프로그램
    849 페이지의 표45.

Java SQLJ 샘플 프로그램
    850 페이지의 표46.

SQL Procedure 샘플 프로그램
    851 페이지의 표47.

ActiveX Data Object, Remote Data Object 및 Microsoft Transaction Server 샘플 프로그램
    853 페이지의 표48.

OLE(Object Linking and Embedding) 자동화 샘플 프로그램
    854 페이지의 표49.

OLE DB(Object Linking and Embedding Database) 테이블 함수
    855 페이지의 표50.

명령행 처리기(CLP) 샘플 프로그램
    855 페이지의 표51.

로그 관리 User Exit 프로그램
856 페이지의 표52.

주:

1. 844 페이지의 표41에는 DB2 API와 Embedded SQL문을 포함한 프로그램이 있습니다. 모든 DB2 API 샘플 프로그램에 대해서는 840 페이지의 표40 및 844 페이지의 표41에서 자세한 내용을 참조하십시오. Java SQLJ를 제외한 모든 Embedded SQL 샘플 프로그램에 대해서는 844 페이지의 표41 및 846 페이지의 표42에서 자세한 내용을 참조하십시오.
2. DB2 CLI UDF 프로그램이 없는 UDF 샘플 프로그램은 847 페이지의 표43에 나와 있습니다. 이에 대해서는 848 페이지의 표44에서 자세한 내용을 참조하십시오.

표 38. 언어별 샘플 파일 확장자

언어	디렉토리	Embedded SQL 프로그램	embedded되지 않는 SQL 프로그램
C	samples/c samples/cli(CLI 프로그램)	.sqc	.c
C++	samples/cpp	.sqC (UNIX) .sqx (Windows & OS/2)	.C (UNIX) .cxx (Windows & OS/2)
COBOL	samples/cobol samples/cobol_mf	.sqb	.cbl
JAVA	samples/java	.sqlj	.java
REXX	samples/rexx	.cmd	.cmd

표 39. 프로그램 그룹별 샘플 파일 확장자

샘플 그룹	디렉토리	파일 확장자
ADO, RDO, MTS	samples\ADO\VB (Visual Basic) samples\ADO\VC (Visual C++) samples\RDO samples\MTS	.bas .frm .vbp (Visual Basic) .cpp .dsp .dsw (Visual C++)
CLP	samples/clp	.db2
OLE	samples\ole\msvb (Visual Basic) samples\ole\msvc (Visual C++)	.bas .vbp (Visual Basic) .cpp (Visual C++)
OLE DB	samples\oledb	.db2
SQL 프로시듀어	samples/sqlproc	.db2 .c .sqc (클라이언트 응용프로그램)
User Exit	samples/c	.cad (OS/2) .cadsm (UNIX & Windows) .cdisk (UNIX & Windows) .ctape (UNIX)

주:

디렉토리 분리문자
UNIX의 경우 /입니다. OS/2 및 Windows 플랫폼에서는 \입니다. 테이블에서 Windows 및/또는 OS/2에서만 디렉토리가 사용 가능한 경우를 제외하면 UNIX 분리문자가 사용됩니다.

파일 확장자
확장이 하나만 존재하는 테이블에서 샘플에 대해 제공됩니다.

Embedded SQL 프로그램
프로그램 수행시 Embedded SQL문이 해석되는 REXX Embedded SQL 프로그램을 제외하고는 사전 처리 컴파일을 요구합니다.

IBM COBOL 샘플
AIX, OS/2 및 Windows 32 비트 운영 체제의 경우에만 cobol 서브디렉토리에 제공됩니다.

Micro Focus Cobol 샘플
AIX, HP-UX, OS/2 및 Windows 32 비트 운영 체제의 경우에만 cobol_mf 서브디렉토리에 제공됩니다.

Java 샘플

Java UDF뿐 아니라 JDBC(Java Database Connectivity) 애플릿, 응용프로그램 및 저장 프로시듀어, SQLJ(Embedded SQL for Java) 애플릿, 응용프로그램 및 저장 프로시듀어도 해당합니다. 지원되는 모든 DB2 플랫폼에서는 Java 샘플을 사용할 수 있습니다.

REXX 샘플

AIX, OS/2 및 Windows NT 운영 체제에 대해서만 제공됩니다.

CLP 샘플

SQL문을 실행하는 명령행 처리기 스크립트입니다.

OLE 샘플

Windows 32 비트 운영 체제 전용으로 제공된, Microsoft Visual Basic 및 MicrosoftVisual C++의 OLE(Object Linking and Embedding)용입니다.

ADO, RDO 및 MTS 샘플

Windows 32 비트 운영 체제 전용으로 제공된, Microsoft Visual Basic 및 Microsoft Visual C++의 ActiveX Data Object 샘플과 Microsoft Visual Basic의 Remote Data Object 및 Microsoft Transaction Server 샘플입니다.

User Exit 샘플

데이터베이스 로그 파일을 보관하고 검색하는 데 사용되는 로그 관리 User Exit 프로그램입니다. 이 파일의 확장자를 `.c`로 변경하고 파일을 C 언어 프로그램으로 컴파일해야 합니다.

DB2를 설치한 디렉토리의 `samples` 서브디렉토리에 샘플 프로그램이 있습니다. 지원되는 각 언어에 대한 서브디렉토리가 있습니다. 다음 예는 지원되는 각 플랫폼에서 C 또는 C++로 작성한 샘플을 찾는 방법을 나타냅니다.

- **UNIX 플랫폼의 경우**

  데이터베이스 인스턴스 디렉토리가 있는 `sqllib/samples/c` 아래에 Embedded SQL 및 DB2 API 프로그램의 C 소스 코드가 있습니다. DB2 CLI 프로그램의 C 소스 코드는 `sqllib/samples/cli`에 있습니다. 샘플 테이블의 프로그램

에 대한 자세한 내용은 DB2 인스턴스가 있는 해당 `samples` 서브디렉토리의 `README` 파일을 참조하십시오. `README` 파일에는 이 책에 나열되어 있지 않은 추가 샘플도 있습니다.

- **OS/2 및 Windows 32 비트 운영 체제의 경우**

  DB2 설치 디렉토리 아래의 `%DB2PATH%\samples\c`에 Embedded SQL 및 DB2 API 프로그램의 C 소스 코드가 있습니다. DB2 CLI 프로그램의 C 소스 코드는 `%DB2PATH%\samples\cli`에 있습니다. `%DB2PATH%` 변수에서 DB2가 설치된 위치를 판별합니다. DB2를 설치한 드라이브에 따라 `%DB2PATH%`는 *drive*:`\sqllib`를 지시합니다. 샘플 테이블의 샘플 프로그램에 대한 자세한 내용은 해당 `%DB2PATH%\samples` 서브디렉토리의 `README` 파일을 참조하십시오. `README` 파일에는 이 책에 나열되어 있지 않은 추가 샘플도 있습니다.

사용자의 플랫폼이 837 페이지의 표38에 설명되어 있지 않은 경우 사용자 환경의 고유 내용에 대해서는 *응용프로그램 빌드 안내서*에서 자세한 내용을 참조하십시오.

샘플 프로그램 디렉토리는 대부분의 플랫폼에서 보통 읽기 전용입니다. 샘플 프로그램을 변경하거나 빌드하기 전에 그 프로그램을 작업 디렉토리로 복사하십시오.

Embedded SQL 샘플이 없는 DB2 API

표 40. Embedded SQL 없는 DB2 API 샘플 프로그램

샘플 프로그램	포함된 API
backrest	• sqlbftcq - 테이블 공간 컨테이너 조회 페치(fetch) • sqlbstsc - 테이블 공간 컨테이너 설정 • sqlfudb - 데이터베이스 구성 갱신 • sqlubkp - 데이터베이스 백업 • sqluroll - 데이터베이스 롤 포워드 • sqlurst - 데이터베이스 복원
checkerr	• sqlaintp - 오류 메시지 확보 • sqlogstt - SQLSTATE 메시지 확보
cli_info	• sqleqryi - 클라이언트 정보 조회 • sqleseti - 클라이언트 정보 설정

표 40. Embedded SQL 없는 DB2 API 샘플 프로그램 (계속)

샘플 프로그램	포함된 API
client	• sqleqryc - 클라이언트 조회 • sqlesetc - 클라이언트 설정
d_dbconf	• sqleatin - 접속 • sqledtin - 분리 • sqlfddb - 데이터베이스 구성 기본값 확보
d_dbmcon	• sqleatin - 접속 • sqledtin - 분리 • sqlfdsys - 데이터베이스 관리 프로그램 구성 기본값 확보
db_udcs	• sqleatin - 접속 • sqlecrea - 데이터베이스 작성 • sqledrpd - 데이터베이스 제거
db2mon	• sqleatin - 접속 • sqlmon - 모니터 스위치 확보/갱신 • sqlmonss - 스냅샷 확보 • sqlmonsz - sqlmonss() 출력 버퍼의 필수 크기 추정 • sqlmrset - 모니터 재설정
dbcat	• sqlecadb - 데이터베이스 카탈로그화 • sqledcls - 데이터베이스 디렉토리 스캔 닫기 • sqledgne - 다음 데이터베이스 디렉토리 항목 확보 • sqledosd - 데이터베이스 디렉토리 스캔 열기 • sqleuncd - 데이터베이스 카탈로그화 해제
dbcmt	• sqledcgd - 데이터베이스 주석 변경 • sqledcls - 데이터베이스 디렉토리 스캔 닫기 • sqledgne - 다음 데이터베이스 디렉토리 항목 확보 • sqledosd - 데이터베이스 디렉토리 스캔 열기 • sqleisig - 신호 핸들러 설치

표 40. Embedded SQL 없는 DB2 API 샘플 프로그램 (계속)

샘플 프로그램	포함된 API
dbconf	• sqleatin - 접속 • sqlecrea - 데이터베이스 작성 • sqledrpd - 데이터베이스 제거 • sqlfrdb - 데이터베이스 구성 재설정 • sqlfudb - 데이터베이스 구성 갱신 • sqlfxdb - 데이터베이스 구성 확보
dbinst	• sqleatcp - 접속 및 암호 변경 • sqleatin - 접속 • sqledtin - 분리 • sqlegins - 인스턴스 확보
dbmconf	• sqleatin - 접속 • sqledtin - 분리 • sqlfrsys - 데이터베이스 관리 프로그램 구성 재설정 • sqlfusys - 데이터베이스 관리 프로그램 구성 갱신 • sqlfxsys - 데이터베이스 관리 프로그램 구성 확보
dbsnap	• sqleatin - 접속 • sqlmonss - 스냅샷 확보
dbstart	• sqlepstart - 데이터베이스 관리 프로그램 시작
dbstop	• sqlefrce - 응용프로그램 강제 • sqlepstp - 데이터베이스 관리 프로그램 중지
dcscat	• sqlegdad - DCS 데이터베이스 카탈로그화 • sqlegdcl - DCS 디렉토리 스캔 닫기 • sqlegdel - 데이터베이스 카탈로그화 해제 • sqlegdge - 데이터베이스의 DCS 디렉토리 항목 확보 • sqlegdgt - DCS 디렉토리 항목 확보 • sqlegdsc - DCS 디렉토리 스캔 열기

표 40. Embedded SQL 없는 DB2 API 샘플 프로그램 (계속)

샘플 프로그램	**포함된 API**
dmscont	• sqleatin - 접속 • sqlecrea - 데이터베이스 작성 • sqledrpd - 데이터베이스 제거
ebcdicdb	• sqleatin - 접속 • sqlecrea - 데이터베이스 작성 • sqledrpd - 데이터베이스 제거
migrate	• sqlemgdb - 데이터베이스 이주
monreset	• sqleatin - 접속 • sqlmrset - 모니터 재설정
monsz	• sqleatin - 접속 • sqlmonss - 스냅샷 확보 • sqlmonsz - sqlmonss() 출력 버퍼의 필수 크기 추정
nodecat	• sqlectnd - 노드 카탈로그화 • sqlencls - 노드 디렉토리 스캔 닫기 • sqlengne - 다음 노드 디렉토리 항목 확보 • sqlenops - 노드 디렉토리 스캔 열기 • sqleuncn - 노드 카탈로그화 해제
restart	• sqlerstd - 데이터베이스 재시작
setact	• sqlesact - 계정 문자열 설정
setrundg	• sqlesdeg - 런타임 등급 설정
sws	• sqleatin - 접속 • sqlmon - 모니터 스위치 확보/갱신
utilapi	• sqlaintp - 오류 메시지 확보 • sqlogstt - SQLSTATE 메시지 확보

DB2 API Embedded SQL 샘플

표 41. DB2 API Embedded SQL 샘플 프로그램

샘플 프로그램	포함된 API
asynrlog	• sqlurlog - 로그 비동기 읽기
autocfg	• db2AutoConfig -- Autoconfig • db2AutoConfigMemory -- Autoconfig Free Memory • sqlfudb -- Update Database Configuration • sqlfusys -- Update Database Manager Configuration • sqlesetc -- Set Client • sqlaintp -- SQLCA Message
dbauth	• sqluadau - 권한 부여
dbstat	• sqlureot - 테이블 재구성 • sqlustat - Runstats
expsamp	• sqluexpr - 내보내기 • sqluimpr - 가져오기
impexp	• sqluexpr - 내보내기 • sqluimpr - 가져오기
loadqry	• db2LoadQuery - 조회 로드
makeapi	• sqlabndx - 바인드 • sqlaprep - 프로그램 사전 처리 컴파일 • sqlepstp - 데이터베이스 관리 프로그램 중지 • sqlepstr - 데이터베이스 관리 프로그램 시작
rebind	• sqlarbnd - 리바인드
rechist	• sqlubkp - 데이터베이스 백업 • sqluhcls - 복구 실행기록 파일 스캔 닫기 • sqluhgne - 다음 복구 실행기록 파일 항목 확보 • sqluhops - 복구 실행기록 파일 스캔 열기 • sqluhprn - 복구 실행기록 파일 제거 • sqluhupd - 복구 실행기록 파일 갱신

표 41. DB2 API Embedded SQL 샘플 프로그램 (계속)

샘플 프로그램	포함된 API
tabscont	• sqlbctcq - 테이블 공간 컨테이너 조회 닫기 • sqlbftcq - 테이블 공간 컨테이너 조회 페치(fetch) • sqlbotcq - 테이블 공간 컨테이너 조회 열기 • sqlbtcq - 테이블 공간 컨테이너 조회 • sqlefmem - 메모리 해제
tabspace	• sqlbctsq - 테이블 공간 조회 닫기 • sqlbftpq - 테이블 공간 조회 페치(fetch) • sqlbgtss - 테이블 공간 통계 확보 • sqlbmtsq - 테이블 공간 조회 • sqlbotsq - 테이블 공간 조회 열기 • sqlbstpq - 단일 테이블 공간 조회 • sqlefmem - 메모리 해제
tload	• sqluexpr - 내보내기 • sqluload - 로드 • sqluvqdp - 테이블의 테이블 공간 Quiesce
tspace	• sqlbctcq - 테이블 공간 컨테이너 조회 닫기 • sqlbctsq - 테이블 공간 조회 닫기 • sqlbftcq - 테이블 공간 컨테이너 조회 페치(fetch) • sqlbftpq - 테이블 공간 조회 페치(fetch) • sqlbgtss - 테이블 공간 통계 확보 • sqlbmtsq - 테이블 공간 조회 • sqlbotcq - 테이블 공간 컨테이너 조회 열기 • sqlbotsq - 테이블 공간 조회 열기 • sqlbstpq - 단일 테이블 공간 조회 • sqlbstsc - 테이블 공간 컨테이너 설정 • sqlbtcq - 테이블 공간 컨테이너 조회 • sqlefmem - 메모리 해제

표 41. DB2 API Embedded SQL 샘플 프로그램 (계속)

샘플 프로그램	포함된 API
utilemb	• sqlaintp - 오류 메시지 확보 • sqlogstt - SQLSTATE 메시지 확보

DB2 API가 없는 Embedded SQL 샘플

표 42. API가 없는 Embedded SQL 샘플

샘플 프로그램 이름	프로그램 설명
adhoc	동적 SQL 명령을 대화식으로 처리하는 동적 SQL 및 SQLDA 구조의 데모를 나타냅니다. SQL 명령은 사용자가 입력하면 그 SQL 명령에 해당하는 출력이 리턴됩니다. 세부사항은 174 페이지의 『예: ADHOC 프로그램』의 내용을 참조하십시오.
advsql	CASE, CAST 및 스칼라 전체 선택과 같은 고급 SQL 표현식 사용의 데모를 나타냅니다.
blobfile	샘플 데이터베이스의 2진 대형 오브젝트(BLOB) 값을 읽고 그 값을 파일에 지정하여 BLOB를 조작하는 데모를 나타냅니다. 외부 표시기를 사용하여 이 파일의 내용을 표시할 수 있습니다.
columns	동적 SQL을 사용하여 처리한 커서 사용의 데모를 나타냅니다. 이 프로그램은 SYSCAT.COLUMNS의 결과 세트를 원하는 스키마 이름 밑에 나열합니다.
cursor	정적 SQL을 사용한 커서 사용의 데모를 나타냅니다. 세부사항은 99 페이지의 『예: 커서 프로그램』의 내용을 참조하십시오.
delet	데이터베이스에서 항목을 삭제하는 정적 SQL의 데모를 나타냅니다.
dynamic	동적 SQL을 사용한 커서 사용의 데모를 나타냅니다.
joinsql	고급 SQL 조인 표현식을 사용한 설명
largevol	파티션된 환경에서의 병렬 조회 처리 및 NFS 파일 시스템을 사용한 결과 세트 병합 자동화의 데모를 나타냅니다. AIX에서만 사용 가능합니다. 세부사항은 630 페이지의 『예: 대용량의 데이터 추출(largevol.c)』의 내용을 참조하십시오.
lobeval	LOB 위치 지정자 사용 및 실제 LOB 데이터 평가 지연의 데모를 나타내냅니다. 세부사항은 395 페이지의 『샘플 LOBEVAL 프로그램 작업 방법』의 내용을 참조하십시오.
lobfile	LOB 파일 핸들 사용의 데모를 나타냅니다. 세부사항은 403 페이지의 『샘플 LOBFILE 프로그램 작업 방법』의 내용을 참조하십시오.
lobloc	LOB 위치 지정자 사용의 데모를 나타냅니다. 세부사항은 388 페이지의 『샘플 LOBLOC 프로그램 작업 방법』의 내용을 참조하십시오.
lobval	LOB 사용의 데모를 나타냅니다.
openftch	정적 SQL을 사용한 행 페치, 갱신 및 삭제의 데모를 나타냅니다. 세부사항은 109 페이지의 『OPENFTCH 프로그램 작업 방법』의 내용을 참조하십시오.

표 42. API가 없는 Embedded SQL 샘플 (계속)

샘플 프로그램 이름	프로그램 설명
recursql	고급 SQL 순환 조회 사용의 데모를 나타냅니다.
sampudf	테이블 항목을 수정하기 위해 구현된 사용자 정의 유형(UDT)와 사용자 정의 함수(UDF)를 설명합니다. 이 프로그램에서 선언된 모든 UDF는 소스화된 UDF입니다.
spclient	spserver 공유 라이브러리에서 저장 프로시듀어를 호출하는 클라이언트 응용프로그램.
spcreate.db2	CREATE PROCEDURE문을 포함하여 spserver 프로그램에서 작성된 저장 프로그램을 등록하는 CLP 스크립트.
spdrop.db2	spserver 프로그램에서 작성된 저장 프로시듀어를 참조 해제할 때 필요한 DROP PROCEDURE문이 들어 있는 CLP 스크립트.
spserver	저장 프로시듀어를 보여주는 서버 프로그램. 클라이언트 프로그램은 spclient입니다.
static	정적 SQL을 설명하여 정보를 검색합니다. 세부사항은 77 페이지의『예: 정적 SQL 프로그램』의 내용을 참조하십시오.
tabsql	고급 SQL 테이블 표현식 사용의 데모를 나타냅니다.
tbdefine	테이블 작성 및 제거를 보여줍니다.
thdsrver	스레드 작성 및 관리를 위해 POSIX 스레드 API를 사용하는 데모를 나타냅니다. 프로그램에서 문맥 풀을 유지보수합니다. generate_work 함수가 메인에서 실행되어 작업자 스레드에서 실행하는 동적 SQL문을 작성합니다. 문맥이 사용 가능할 경우 스레드가 작성되며, 지정된 작업을 실행하도록 디스패치됩니다. 생성된 작업은 sample 데이터베이스의 STAFF 또는 EMPLOYEE 테이블에서 항목을 삭제하는 명령문으로 구성됩니다. 이 프로그램은 UNIX 플랫폼에서만 사용할 수 있습니다.
trigsql	고급 SQL 트리거 및 제한조건을 사용하는 예
udfcli	udfsrv 프로그램에서 작성되고 sample 데이터베이스에 있는 테이블을 액세스하기 위해 서버에 저장된 사용자 정의 함수(UDF)를 호출하는 방법을 보여줍니다.
updat	Demonstrates static SQL to update a database. 세부사항은 122 페이지의『예: UPDAT 프로그램』의 내용을 참조하십시오.
varinp	매개변수 표시문자를 사용한 Embedded Dynamic SQL문 호출에 대한 변수 입력의 데모를 나타냅니다. 세부사항은 181 페이지의『VARINP 프로그램 작업 방법』의 내용을 참조하십시오.

사용자 정의 함수 샘플

표 43. 사용자 정의 함수 샘플 프로그램

샘플 프로그램 이름	프로그램 설명
DB2Udf.java	정수 나눔, 문자 대형 오브젝트(CLOB)의 조작 및 Java 인스턴스 변수 사용 등 몇 가지 타스크의 데모를 나타내는 Java UDF

표 43. 사용자 정의 함수 샘플 프로그램 (계속)

샘플 프로그램 이름	프로그램 설명
udfsrv.c	샘플 데이터베이스 테이블을 액세스하기 위해 사용자 정의 함수 ScalarUDF와 함께 라이브러리를 작성합니다.
UDFsrv.java	Java 사용자 정의 함수 사용의 데모를 나타냅니다.

DB2 Call Level Interface 샘플

표 44. DB2 Universal Database의 샘플 CLI 프로그램

샘플 프로그램 이름	프로그램 설명
명령 유틸리티 파일	
utilcli.c	CLI 샘플은 유틸리티 기능을 사용합니다.
utilapi.c	DB2 API를 호출하는 Utility 기능.
응용프로그램 레벨 - DB2와 CLI 응용프로그램 레벨을 다루는 샘플.	
apinfo.c	응용프로그램 레벨 정보 확보 및 설정 방법
aphndls.c	핸들 할당 및 해제 방법
apsqlca.c	SQLCA 데이터 사용 방법.
설치 이미지 레벨 - DB2 및 CLI의 설치 이미지 레벨을 다루는 샘플.	
ilinfo.c	설치 레벨 정보 확보(예: CLI 드라이버의 버전) 및 설정 방법
인스턴스 레벨 - DB2 및 CLI의 인스턴스 레벨을 다루는 샘플.	
ininfo.c	인스턴스 레벨 정보 확보 및 설정 방법
데이터베이스 레벨 - DB2에 있는 데이터베이스 오브젝트를 다루는 샘플.	
dbconn.c	데이터베이스로 부터 연결해제 및 연결 방법.
dbinfo.c	데이터베이스 레벨에서 정보를 확보하고 설정하는 방법.
dbmconn.c	여러 데이터베이스에서 연결 및 연결해제하는 방법(DB2 API를 사용하여 두 번째 데이터베이스를 작성하고 제거함).
dbmuse.c	여러 데이터베이스를 사용하여 트랜잭션을 수행하는 방법(DB2 API를 사용하여 두 번째 데이터베이스를 작성하고 제거함).
dbnative.c	ODBC escape절이 들어 있는 명령문을 데이터 소스 고유 형식으로 변환하는 방법.
dbuse.c	데이터베이스 오브젝트로 작업 방법.
dbusemx.sqc	단일 데이터베이스를 embedded SQL과 함께 사용하는 방법
테이블 레벨 - DB2에서 테이블 오브젝트를 다루는 샘플.	
tbconstr.c	테이블 제한조건과 작업 방법.
tbconstr.c	테이블 제거 및 변경, 작성 방법.

표 44. DB2 Universal Database의 샘플 CLI 프로그램 (계속)

샘플 프로그램 이름	프로그램 설명
tbinfo.c	테이블 레벨에서 정보를 확보하고 설정하는 방법.
tbmod.c	테이블에 있는 정보 수정 방법.
tbread.c	테이블에 있는 정보 읽는 방법.
데이터 유형 레벨 - 데이터 유형을 다루는 샘플	
dtinfo.c	데이터 유형에 정보를 받는 방법.
dtlob.c	LOB 데이터 쓰기 및 읽는 방법.
dtudt.c	사용자 정의 구별 유형 제거 및 사용, 작성 방법.
UDF 레벨 - 사용자 정의 함수를 보여주는 샘플	
udfcli.c	udfsrv.c에서 사용자 정의 함수를 호출하는 클라이언트 응용프로그램.
udfsrv.c	udfcli.c 샘플에서 호출된 사용자 정의 함수.
저장 프로시듀어 레벨 - CLI에서 저장 프로시듀어를 보여주는 샘플.	
spcreate.db2	CREATE PROCEDURE문을 실행하는 CLP 스크립트.
spdrop.db2	카탈로그에서 저장 프로시듀어를 제거하는 CLP 스크립트.
spclient.c	spserver.c에서 선언된 서버 함수를 호출할 때 사용되는 클라이언트 프로그램.
spserver.c	서버에서 구축되고 실행되는 저장 프로시듀어 함수.
spcall.c	저장 프로시듀어를 호출하는 프로그램.

주: samples/cli 디렉토리의 기타 파일은 다음과 같습니다.

- README - 모든 예제 파일을 나열합니다.
- makefile - 모든 파일의 Makefile
- 응용프로그램 및 저장 프로시듀어용 파일을 빌드합니다.

Java 샘플

표 45. JDBC(Java Database Connectivity) 샘플 프로그램

샘플 프로그램 이름	프로그램 설명
DB2Appl.java	사용자 특권을 호출하여 샘플 데이터베이스를 조회하는 JDBC 응용프로그램
DB2Applt.java	JDBC 애플릿 드라이버를 사용하여 데이터베이스를 조회하는 JDBC 애플릿. 이 애플릿은 DB2Applt.html에 지정된 사용자 이름, 암호, 서버 및 포트 번호를 사용합니다.
DB2Applt.html	애플릿 샘플 프로그램 DB2Applt를 Embed하는 HTML 파일. 이 파일은 서버 및 사용자 정보로 사용자 정의해야 합니다.
DB2UdCli.java	사용자 정의 함수 DB2Udf를 호출하는 Java 클라이언트 응용프로그램
Dynamic.java	정적 SQL을 사용한 커서 사용의 데모를 나타냅니다.

표 45. JDBC(Java Database Connectivity) 샘플 프로그램 (계속)

샘플 프로그램 이름	프로그램 설명
MRSPcli.java	서버 프로그램 MRSPsrv를 호출하는 클라이언트 프로그램입니다. 이 프로그램은 Java 저장 프로시듀어에서 리턴 중인 복수 결과 세트의 데모를 나타냅니다.
MRSPsrv.java	클라이언트 프로그램 MRSPcli에서 호출하는 서버 프로그램입니다. 이 프로그램은 Java 저장 프로시듀어에서 리턴 중인 복수 결과 세트의 데모를 나타냅니다.
Outcli.java	SQLJ 저장 프로시듀어 Outsrv를 호출하는 Java 클라이언트 응용프로그램
PluginEx.java	DB2 웹 제어 센터에 새로운 메뉴 항목과 도구 막대 버튼을 추가하는 Java 프로그램
Spclient.java	Spserver 저장 프로시듀어 클래스에 있는 PARAMETER STYLE JAVA 저장 프로시듀어를 호출하는 JDBC 클라이언트 응용프로그램.
Spcreate.db2	Spserver 클래스에 들어 있는 메소드를 저장 프로시듀어로 등록하는 CREATE PROCEDURE문이 들어 있는 CLP 스크립트.
Spdrop.db2	Spserver 클래스에 들어 있는 저장 프로시듀어를 참조 해제하는 데 필요한 DROP PROCEDURE문이 들어 있는 CLP 스크립트.
Spserver.java	PARAMETER STYLE JAVA 저장 프로시듀어를 보여주는 JDBC 프로그램. 클라이언트 프로그램은 Spclient.java 입니다.
UDFcli.java	UDFsrv Java 사용자 정의 함수 라이브러리에 있는 함수를 호출하는 JDBC 클라이언트 응용프로그램.
UseThrds.java	스레드를 사용하여 SQL문을 비동기적으로 수행하는 방법을 나타냅니다. CLI 샘플 async.c의 JDBC 버전입니다.
V5SpCli.java	DB2GENERAL 저장 프로시듀어 V5Stp.java를 호출하는 Java 클라이언트 응용프로그램
V5Stp.java	서버에서 EMPLOYEE 테이블을 갱신하고 클라이언트로 새로운 급여와 급료 지불 명부 정보를 리턴하는 저장 프로시듀어. 클라이언트 프로그램은 V5SpCli.java입니다.
Varinp.java	매개변수 표시문자를 사용한 Embedded Dynamic SQL문 호출에 대한 변수 입력의 데모를 나타냅니다.

표 46. SQLJ(Embedded SQL for Java) 샘플 프로그램

샘플 프로그램 이름	프로그램 설명
App.sqlj	정적 SQL을 사용하여 샘플 데이터베이스의 EMPLOYEE 테이블에서 데이터를 검색하고 갱신합니다.
Applt.sqlj	JDBC 애플릿 드라이버를 사용하여 데이터베이스를 조회하는 애플릿. 이 애플릿은 Applt.html에 지정된 사용자 이름, 암호, 서버 및 포트 번호를 사용합니다.
Applt.html	애플릿 샘플 프로그램 Applt를 Embed하는 HTML 파일. 이 파일은 서버 및 사용자 정보로 사용자 정의해야 합니다.
Cursor.sqlj	정적 SQL을 사용하여 반복자의 데모를 나타냅니다.
OpF_Curs.sqlj	Openftch 프로그램의 클래스 파일
Openftch.sqlj	정적 SQL을 사용한 행 페치, 갱신 및 삭제의 데모를 나타냅니다.

표 46. SQLJ(Embedded SQL for Java) 샘플 프로그램 (계속)

샘플 프로그램 이름	프로그램 설명
Outsrv.sqlj	SQLDA 구조를 사용한 저장 프로시듀어의 데모를 나타냅니다. 이 저장 프로시듀어는 sample 데이터베이스의 STAFF 테이블에 있는 직원의 중간 급여로 SQLDA를 채웁니다. 데이터베이스 처리(중앙값 찾기) 후 저장 프로시듀어는 채워진 SQLDA와 SQLCA 상태를 JDBC 클라이언트 응용프로그램, 즉 Outcli로 리턴합니다.
Stclient.sqlj	Stserver SQLJ 저장 프로시듀어 프로그램에 의해 작성된 PARAMETER STYLE JAVA 저장 프로시듀어를 호출하는 SQLJ 클라이언트 응용프로그램.
Stcreate.db2	Stserver 클래스에 들어 있는 메소드를 저장 프로시듀어로 등록하는 CREATE PROCEDURE 문이 들어 있는 CLP 스크립트.
Stdrop.db2	Stserver 클래스에 들어 있는 저장 프로시듀어를 참조 해제하는 데 필요한 DROP PROCEDURE문이 들어 있는 CLP 스크립트.
Stserver.sqlj	PARAMETER STYLE JAVA 저장 프로시듀어를 보여주는 SQLJ 프로그램. 클라이언트 프로그램은 Stclient.sqlj입니다.
Static.sqlj	정적 SQL을 사용하여 정보를 검색합니다.
Stp.sqlj	서버에서 EMPLOYEE 테이블을 갱신하고 새 급여 및 급료 정보를 JDBC 클라이언트 프로그램, 즉 StpCli에 리턴하는 저장 프로시듀어.
UDFclie.sqlj	UDFsrv Java 사용자 정의 함수 라이브러리에서 함수를 호출하는 클라이언트 응용프로그램.
Updat.sqlj	정적 SQL을 사용하여 데이터베이스를 갱신합니다.

SQL 프로시듀어 샘플

표 47. SQL 프로시듀어 샘플 프로그램

샘플 프로그램 이름	프로그램 설명
basecase.db2	UPDATE_SALARY 프로시듀어는 ″샘플″ 데이터베이스의 ″staff″ 테이블에서 ″empno″ IN 매개변수로 식별되는 직원의 급여를 올려줍니다. 프로시듀어는 ″rating″ IN 매개변수를 사용하는 CASE문에 따라 인상을 결정합니다.
basecase.sqc	UPDATE_SALARY 프로시듀어를 호출합니다.
baseif.db2	UPDATE_SALARY_IF 프로시듀어는 ″샘플″ 데이터베이스의 ″staff″ 테이블에서 ″empno″ IN 매개변수로 식별되는 직원의 급여를 올려줍니다. 프로시듀어는 ″rating″ IN 매개변수를 사용하는 IF문에 따라 인상을 결정합니다.
baseif.sqc	UPDATE_SALARY_IF 프로시듀어를 호출합니다.
dynamic.db2	CREATE_DEPT_TABLE 프로시듀어는 동적 DDL을 사용하여 새 테이블을 작성합니다. 테이블 이름은 프로시듀어에 대한 IN 매개변수의 값을 기반으로 합니다.
dynamic.sqc	CREATE_DEPT_TABLE 프로시듀어를 호출합니다.

표 47. SQL 프로시듀어 샘플 프로그램 (계속)

샘플 프로그램 이름	프로그램 설명
iterate.db2	ITERATOR 프로시듀어는 FETCH 루프를 사용하여 "department" 테이블에서 데이터를 검색합니다. "deptno" 컬럼의 값이 "D11"이 아니면 수정된 데이터가 "department" 테이블에 삽입됩니다. "deptno" 컬럼의 값이 "D11"이면 ITERATE문은 제어 흐름을 다시 LOOP문의 시작으로 전달합니다.
iterate.sqc	ITERATOR 프로시듀어를 호출합니다.
leave.db2	LEAVE_LOOP 프로시듀어는 "not_found" 조건 핸들러가 LEAVE문을 호출하기 전에 LOOP문에서 수행된 FETCH 조작의 수를 계산합니다. LEAVE문으로 제어 흐름은 루프를 종료시키고 저장 프로시듀어를 완료합니다.
leave.sqc	LEAVE_LOOP 프로시듀어를 호출합니다.
loop.db2	LOOP_UNTIL_SPACE 프로시듀어는 커서가 "midinit" 컬럼에 대해 공백(' ') 값의 행을 검색할 때까지 LOOP문에서 수행된 FETCH 조작의 수를 계산합니다. 루프 문으로 제어 흐름은 루프를 종료시키고 저장 프로시듀어를 완료합니다.
loop.sqc	LOOP_UNTIL_SPACE 프로시듀어를 호출합니다.
nestcase.db2	BUMP_SALARY 프로시듀어는 중첩된 CASE문을 사용하여 "샘플" 데이터베이스의 "staff" 테이블에서 dept IN 매개변수로 식별되는 부서에 있는 직원의 급여를 올려줍니다.
nestcase.sqc	BUMP_SALARY 프로시듀어를 호출합니다.
nestif.db2	BUMP_SALARY_IF 프로시듀어는 중첩된 IF문을 사용하여 "샘플" 데이터베이스의 "staff" 테이블에서 dept IN 매개변수로 식별되는 부서에 있는 직원의 급여를 올려줍니다.
nestif.sqc	BUMP_SALARY_IF 프로시듀어를 호출합니다.
repeat.db2	REPEAT_STMT 프로시듀어는 커서가 더 이상 행을 검색할 수 없을 때까지 반복문에서 수행한 FETCH 조작의 수를 계산합니다. 조건 핸들러로 제어 흐름은 반복 루프를 종료시키고 저장 프로시듀어를 완료합니다.
repeat.sqc	REPEAT_STMT 프로시듀어를 호출합니다.
rsultset.c	MEDIAN_RESULT_SET 프로시듀어를 호출하고 중앙 급여를 표시하며, SQL 프로시듀어에서 생성된 결과 세트를 표시합니다. 이 클라이언트는 결과 세트를 승인할 수 있는 CLI API를 사용하여 작성됩니다.
rsultset.db2	MEDIAN_RESULT_SET 프로시듀어는 "샘플" 데이터베이스의 "staff" 테이블에서 "dept" IN 매개변수로 식별된 부서에서 직원의 급여를 확보합니다. 중앙값은 급여 OUT 매개변수에 할당되고 "rsultset" 클라이언트로 리턴됩니다. 프로시듀어는 WITH RETURN 커서를 열어서 급여가 중앙값보다 큰 직원의 결과 세트를 리턴합니다. 프로시듀어는 결과 세트를 클라이언트에게 리턴합니다.
spserver.db2	이 CLP에 있는 SQL 프로시듀어는 기본 오류 처리, 중첩 저장 프로시듀어 호출 그리고 클라이언트 응용프로그램이나 호출하는 응용프로그램에 결과 세트를 리턴하는 작업을 보여줍니다. CLI 샘플 디렉토리에서 "spcall" 응용프로그램을 사용하여 프로시듀어를 호출할 수 있습니다. C 및 CPP 샘플 디렉토리에서 "spclient" 응용프로그램을 사용하여 결과 세트를 리턴하지 않는 프로시듀어를 호출할 수도 있습니다.

표 47. SQL 프로시듀어 샘플 프로그램 (계속)

샘플 프로그램 이름	프로그램 설명
whiles.db2	DEPT_MEDIAN 프로시듀어는 "샘플" 데이터베이스의 "staff" 테이블에서 "dept" IN 매개변수로 식별되는 부서에서 직원의 중간 급여를 확보합니다. 중앙값은 급여 OUT 매개변수에 할당되고 "whiles" 클라이언트로 리턴됩니다. whiles 클라이언트는 중간 급여를 인쇄합니다.
whiles.sqc	DEPT_MEDIAN 프로시듀어를 호출합니다.

ADO, RDO 및 MTS 샘플

표 48. ADO, RDO 및 MTS 샘플 프로그램

샘플 프로그램 이름	프로그램 설명
Bank.vbp	사용자 계정에 대한 트랜잭션을 수행하는 기능으로 은행 지점의 데이터를 작성하고 유지보수하는 RDO 프로그램. 응용프로그램에서 데이터를 저장하는 데 필요한 테이블을 작성하는 DDL이 있으므로 프로그램은 사용자가 지정한 데이터베이스를 사용할 수 있습니다.
Blob.vbp	이 ADO 프로그램은 BLOB 데이터 검색의 데모를 나타냅니다. 이 프로그램은 sample 데이터베이스의 emp_photo 테이블에서 그림을 검색하여 표시합니다. 또한 지역 파일의 이미지로 emp_photo 테이블의 이미지를 대체할 수도 있습니다.
BLOBAccess.dsw	이 샘플은 Microsoft Visual C++를 사용하여 ADO/Blob 액세스를 강조표시하는 데모를 나타냅니다. Visual Basic 샘플 Blob.vbp와 비슷합니다. BLOB 샘플에는 두 가지 주요 함수가 있습니다. 1. Sample 데이터베이스에서 BLOB를 읽고 화면으로 표시합니다. 2. 파일에서 BLOB를 읽고 데이터베이스에 삽입합니다.(가져오기)
Connect.vbp	이 ADO 프로그램에서는 연결 오브젝트를 작성하여 sample 데이터베이스에 대한 연결을 설정합니다. 일단 완료하면 프로그램에서 연결해제하고 나갑니다.
Commit.vbp	이 응용프로그램에서는 ADO의 자동 확약/수작업 확약 기능 사용의 데모를 나타냅니다. 이 프로그램은 sample 데이터베이스의 EMPLOYEE 테이블에서 직원 번호와 이름을 조회합니다. 자동 확약 또는 수작업 확약 모드로 데이터베이스에 연결하는 옵션이 있습니다. 자동 확약 모드에서는 데이터베이스에서 레코드에 대한 사용자의 모든 변경사항이 자동으로 갱신됩니다. 수작업 확약 모드에서는 레코드의 내용을 변경하기 전에 사용자가 트랜잭션을 시작해야 할 수 있습니다. 트랜잭션 시작 후의 변경사항은 구간 복원을 수행하여 실행 취소할 수 있습니다. 트랜잭션을 확약하여 변경사항을 영구적으로 저장할 수 있습니다. 프로그램을 나가면 자동으로 변경사항이 구간 복원됩니다.

표 48. ADO, RDO 및 MTS 샘플 프로그램 (계속)

샘플 프로그램 이름	프로그램 설명
db2com.vbp	Visual Basic 프로젝트는 Microsoft Transaction Server를 사용하여 데이터베이스를 갱신하는 작업을 보여줍니다. 이는 db2mts.vbp 클라이언트 프로그램에서 사용되는 서버 DLL을 작성하고 4 가지 클래스 모듈이 있습니다. • UpdateNumberColumn.cls • UpdateRow.cls • UpdateStringColumn.cls • VerifyUpdate.cls 이 프로그램의 경우 DB2MTS 임시 테이블이 sample 데이터베이스에 작성됩니다.
db2mts.vbp	이는 Microsoft Transaction Server를 사용하여 db2com.vbp에서 작성된 서버 DLL을 호출하는 클라이언트 프로그램의 Visual Basic 프로젝트입니다.
Select-Update.vbp	이 ADO 프로그램은 Connect.vbp와 같은 기능을 수행하지만 GUI 인터페이스도 제공합니다. 이 인터페이스를 사용하여 sample 데이터베이스의 ORG 테이블에 저장된 데이터를 보고, 갱신하고, 삭제할 수 있습니다.
Sample.vbp	이 Visual Basic 프로젝트는 ADO를 통해 Keyset 커서를 사용하여 sample 데이터베이스에 있는 모든 데이터에 그래픽 사용자 인터페이스를 제공합니다.
VarCHAR.dsp	ADO를 사용하여 VarChar 데이터를 텍스트 필드로 액세스하는 Visual C++ 프로그램. 이는 그래픽 사용자 인터페이스를 제공하여 sample 데이터베이스의 ORG 테이블에서 데이터를 보고 갱신할 수 있게 합니다.

오브젝트 링크 및 샘플 포함

표 49. OLE(Object Linking and Embedding) 샘플 프로그램

샘플 프로그램 이름	프로그램 설명
sales	Visual Basic으로 구현한 Microsoft Excel 판매 스프레드시트에 대한 롤업 조회의 데모를 나타냅니다.
names	Visual Basic으로 구현한 Lotus Notes 주소록을 조회합니다.
inbox	Visual Basic으로 구현한 OLE/Messaging을 통해 Microsoft Exchange 받은 편지함 전자우편 메시지를 조회합니다.
invoice	전자우편 첨부 파일로 Microsoft Word 송장 문서를 전송하는, Visual Basic으로 구현한 OLE 자동화 사용자 정의 함수(UDF)입니다.
bcounter	OLE 자동화 사용자 정의 함수는 인스턴스 변수(Visual Basic으로 구현)를 사용하여 스크래치 패드를 보여줍니다.
ccounter	Visual C++로 구현한 카운터 OLE 자동화 사용자 정의 함수입니다.

표 49. OLE(Object Linking and Embedding) 샘플 프로그램 (계속)

샘플 프로그램 이름	프로그램 설명
salarysrv	Visual Basic으로 구현한 sample 데이터베이스의 STAFF 테이블에서 중간 급여를 계산하는 OLE 자동화 저장 프로시듀어입니다.
salarycltvc	salarysrv Visual Basic 저장 프로시듀어를 호출하는 Visual C++ embedded SQL 샘플.
salarycltvb	salarysrv Visual Basic 저장 프로시듀어를 호출하는 Visual Basic DB2 CLI 샘플.
testcli	tstsrv(Visual C++로 구현) 저장 프로시듀어를 호출하는 OLE 자동화 embedded SQL 클라이언트 응용프로그램.
tstsrv	클라이언트와 저장 프로시듀어(Visual C++로 구현) 간에 다양한 유형을 전달하는 과정을 보여주는 OLE 자동화 저장 프로시듀어.

표 50. OLE DB(Object Linking and Embedding Database) 테이블 함수

샘플 프로그램 이름	프로그램 설명
jet.db2	Microsoft.Jet.OLEDB.3.51 Provider
mapi.db2	MAPI용 INTERSOLV Connect OLE DB
msdaora.db2	Oracle용 Microsoft OLE DB Provider
msdasql.db2	ODBC 드라이버용 Microsoft OLE DB Provider
msidxs.db2	Microsoft OLE DB Index Server Provider
notes.db2	Notes용 INTERSOLV Connect OLE DB
sampprov.db2	Microsoft OLE DB Sample Provider
sqloledb.db2	SQL 서버용 Microsoft OLE DB Provider

명령행 처리기 샘플

표 51. 명령행 처리기(CLP) 샘플 프로그램

샘플 파일 이름	파일 설명
const.db2	CHECK CONSTRAINT절로 테이블을 작성합니다.
cte.db2	공통 테이블 표현식의 데모를 나타냅니다. 이 고급 SQL문의 데모를 나타내는 해당 샘플 프로그램은 tabsql입니다.
flt.db2	순환 조회의 데모를 나타냅니다. 이 고급 SQL문의 데모를 나타내는 해당 샘플 프로그램은 recursql입니다.
join.db2	테이블의 외부 조인의 데모를 나타냅니다. 이 고급 SQL문의 데모를 나타내는 해당 샘플 프로그램은 joinsql입니다.
stock.db2	트리거 사용의 데모를 나타냅니다. 이 고급 SQL문의 데모를 나타내는 해당 샘플 프로그램은 trigsql입니다.

표 51. 명령행 처리기(CLP) 샘플 프로그램 (계속)

샘플 파일 이름	파일 설명
testdata.db2	RAND() 및 TRANSLATE()와 같은 DB2 내장 함수를 사용하여 랜덤하게 생성된 테스트 데이터를 테이블에 상주합니다.
thaisort.db2	이 스크립트는 특히 태국 사용자를 위한 내용입니다. 태국 정렬은 앞에 나오는 모음과 자음의 사전 정렬/스왑핑과 올바른 정렬 순서의 데이터를 보기 위해 사후 정렬을 요구하는 음성학 정렬입니다. 이 파일은 UDF 함수 사전 정렬 및 사후 정렬을 작성하고, 테이블을 작성한 다음 테이블에 대해 함수를 호출하여 테이블 데이터를 정렬합니다. 이 프로그램을 수행하려면 먼저 C 소스 파일 udf.c에서 사용자 정의 함수 프로그램 udf를 빌드해야 합니다.

로그 관리 User Exit 샘플

표 52. 로그 User Exit 샘플 프로그램

샘플 파일 이름	파일 설명
db2uext2.cadsm	이 파일은 ADSM(ADSTAR DSM) API를 활용하여 데이터베이스 로그 파일을 보존하고 검색하는 샘플 User Exit입니다. 이 샘플은 수신된 매개변수와 시간소인을 포함하여 호출의 감사 추적(각 옵션에 대해 별도의 파일에 저장)을 제공합니다. 또한 문제점 판별을 위해 오류 분리 문자열과 시간소인을 포함하여 오류에 호출의 오류 추적도 제공합니다. 이런 옵션을 사용하지 않을 수도 있습니다. 파일의 이름은 db2uext2.c로 바꾸고 C 프로그램으로 컴파일해야 합니다. UNIX 및 Windows 32 비트 운영 체제에서 사용할 수 있습니다. OS/2 버전은 db2uexit.cad입니다.
db2uexit.cad	이 파일은 db2uext2.cadsm의 OS/2 버전입니다. 파일의 이름은 db2uexit.c로 바꾸고 C 프로그램으로 컴파일해야 합니다.
db2uext2.cdisk	프로그램이 있는 특정 플랫폼에 시스템 복사 명령을 활용하는 샘플 User Exit입니다. 이 프로그램은 수신된 매개변수와 시간소인을 포함하여 호출의 감사 추적(각 옵션에 대해 별도의 파일에 저장)을 제공합니다. 또한 문제점 판별을 위해 오류 분리 문자열과 시간소인을 포함하여 오류에 호출의 오류 추적도 제공합니다. 이런 옵션을 사용하지 않을 수도 있습니다. 파일의 이름은 db2uext2.c로 바꾸고 C 프로그램으로 컴파일해야 합니다. UNIX 및 Windows 32 비트 운영 체제에서 사용할 수 있습니다.
db2uext2.ctape	프로그램이 있는 특정 UNIX 플랫폼에 시스템 테이프 명령을 활용하는 샘플 User Exit입니다. 프로그램은 데이터베이스 로그 파일을 보존하고 검색합니다. 시스템 테이프 명령의 모든 제한사항은 이 User Exit의 제한사항이 됩니다. 이 샘플은 수신된 매개변수와 시간소인을 포함하여 호출의 감사 추적(각 옵션에 대해 별도의 파일에 저장)을 제공합니다. 또한 문제점 판별을 위해 오류 분리 문자열과 시간소인을 포함하여 오류에 호출의 오류 추적도 제공합니다. 이런 옵션을 사용하지 않을 수도 있습니다. 파일의 이름은 db2uext2.c로 바꾸고 C 프로그램으로 컴파일해야 합니다. UNIX 플랫폼에서만 사용 가능합니다.

부록C. DB2DARI 및 DB2GENERAL 저장 프로시듀어와 UDF

DB2DARI 저장 프로시듀어 857
클라이언트 응용프로그램에서 SQLDA 사용 858
DB2DARI 클라이언트에서 호스트 변수 사용 858
저장 프로시듀어에서 SQLDA 사용 . . . 859
데이터 구조 조작 859
데이터 구조 사용의 요약 860
입력/출력 SQLDA 및 SQLCA 구조 . . 861
DB2DARI 저장 프로시듀어에 대한 저장 값 862
DB2GENERAL UDF 및 저장 프로시듀어 863
지원되는 SQL 데이터 유형 863
저장 프로시듀어 및 UDF용 클래스 . . . 865
COM.ibm.db2.app.StoredProc . . . 866
COM.ibm.db2.app.UDF 867
COM.ibm.db2.app.Lob 870
COM.ibm.db2.app.Blob 870
COM.ibm.db2.app.Clob 871
비분리 저장 프로시듀어 872
예제 입력-SQLDA 프로그램 873
입력-SQLDA 클라이언트 응용프로그램의 작업 방식 874
C 예: V5SPCLI.SQC 876
예제 입력-SQLDA 저장 프로시듀어의 작업 방식 878
C 예: V5SPSRV.SQC 879

이 장에서는 DB2DARI 및 DB2GENERAL 매개변수 스타일 저장 프로시듀어와 DB2GENERAL UDF 작성 방법에 대해 설명합니다.

DB2DARI 저장 프로시듀어

호출하면 DB2DARI 저장 프로시듀어에서 다음을 수행합니다.

1. 클라이언트 응용프로그램에서 SQLDA 데이터 구조를 수용합니다. 호스트 변수는 SQL CALL문을 실행하면 데이터베이스 관리 프로그램에서 생성하는 SQLDA 데이터 구조를 통해 전달됩니다.
2. 클라이언트 응용프로그램과 동일한 트랜잭션 아래의 데이터베이스 서버에서 실행합니다.
3. SQLCA 정보와 선택적 출력 데이터를 클라이언트 응용프로그램으로 리턴합니다.

© Copyright IBM Corp. 1993, 2000

클라이언트 응용프로그램에서 SQLDA 사용

SQLDA 구조를 사용하여 저장 프로시듀어로 값을 전달하려면 저장 프로시듀어를 호출하기 전에 다음 단계를 수행하십시오.

1. 필수 갯수의 기본 SQLVAR 구성요소가 있는 구조의 저장영역을 할당합니다.
2. SQLN 필드를 할당된 SQLVAR 구성요소의 수로 설정합니다.
3. SQLD 필드를 실제로 사용된 SQLVAR 구성요소의 수로 설정합니다.
4. 다음과 같이 사용된 각 SQLVAR 구성요소를 *초기화*합니다.
   - SQLTYPE 필드를 적절한 데이터 유형으로 설정합니다.
   - SQLLEN 필드를 데이터 유형의 *크기*로 설정합니다.
   - SQLTYPE 및 SQLLEN의 값에 기초한 SQLDATA 및 SQLIND 필드의 저장영역을 할당합니다.

사용자의 응용프로그램에서 2진 데이터용으로 정의된 문자열에 대해 작업할 경우 SQLDAID 필드를 *초기화*하여 SQLDA에 2진 데이터용 정의와 2진 데이터용 구성요소를 정의하는 각 SQLVAR의 SQLNAME 필드가 있음을 나타내야 합니다.

사용자의 응용프로그램에서 대형 오브젝트(LOB) 즉, CLOB, BLOB 또는 DBCLOB 유형의 데이터에 대해 작업할 경우 2차 SQLVAR 구성요소도 *초기화*해야 합니다. SQLDA 구조에 대해서는 *SQL 참조서*에서 자세한 내용을 참조하십시오.

DB2DARI 클라이언트에서 호스트 변수 사용

213 페이지의 『호스트 변수 할당』에서 언급한 것과 같은 접근 방법을 사용하여 SQLVAR를 선언하십시오. 또한 클라이언트 응용프로그램에서 출력 전용 SQLVAR을 859 페이지의 『데이터 구조 조작』에서 언급한 대로 -1로 설정해야 합니다. 그러면 표시기만 전송되므로 SQLDATA 포인터의 내용을 전달할 필요를 없애 매개변수 전달 메카니즘의 성능을 향상시킬 수 있습니다. 이런 매개변수의 경우 SQLTYPE 필드를 널(NULL) 입력 가능 데이터 유형으로 설정해야 합니다. SQLTYPE에서 널 입력 불가능 데이터 유형을 지시할 경우 데이터베이스 관리 프로그램에서 표시기 변수를 점검하지 못합니다.

저장 프로시듀어에서 SQLDA 사용

저장 프로시듀어는 SQL CALL문으로 호출되며 클라이언트 응용프로그램에서 전달하는 데이터를 사용하여 실행됩니다. 정보는 저장 프로시듀어의 SQLDA 구조를 통해 클라이언트 응용프로그램으로 리턴됩니다.

SQL CALL문의 매개변수는 입력 및 출력 매개변수로 처리되며 저장 프로시듀어를 위해 다음 형식으로 변환됩니다.

```
SQL_API_RC SQL_API_FN proc_name( void *reserved1,
                                 void *reserved2,
                                 struct sqlda *inoutsqlda,
                                 struct sqlca *sqlca )
```

SQL_API_FN은 지원되는 각 운영 체제에 따라 달라질 수 있는 함수의 호출 변환을 지정하는 매크로입니다. 저장 프로시듀어 또는 UDF를 작성할 때 이 매크로가 필요합니다.

다음은 CALL문을 사용하여 서버의 매개변수 목록에 맵핑하는 방법의 예입니다.

```
CALL OUTSRV (:empno:empind,:salary:salind)
```

이 호출의 매개변수는 두 SQLDA가 포함된 SQLDA 구조로 변환됩니다. 첫번째 SQLVAR은 empno 호스트 변수와 empind 표시기 변수를 지시합니다. 두 번째 SQLVAR은 salary 호스트 변수와 salind 표시기 변수를 지시합니다.

주: SQLD 구성요소의 수를 0으로 설정한 경우 SQLDA 구조는 저장 프로시듀어로 전달되지 않습니다. 이 경우 SQLDA가 전달되지 않으면 저장 프로시듀어에서 널 포인터를 수신합니다.

데이터 구조 조작

데이터베이스 관리 프로그램에서는 데이터베이스 서버에서 자동으로 중복 SQLDA 구조를 할당합니다. 네트워크 통신량을 줄이려면 입력 전용의 호스트 변수와 출력 전용의 호스트 변수를 지시하는 것이 중요합니다. 클라이언트 프로시듀어에서 출력 전용 SQLVAR의 표시기를 -1로 설정해야 합니다. 서버 프로시듀어에서 입력 전용 SQLVAR의 표시기를 -128로 설정해야 합니다. 그렇게 하면 데이터베이스 관리 프로그램에서 전달된 SQLVAR을 선택할 수 있습니다.

클라이언트 또는 서버에서 표시기 변수를 음의 값으로 설정하면 즉, SQLVAR을 전달하지 못하게 설정하면 표시기 변수는 재설정되지 않습니다. SQLVAR에서 참조하는 호스트 변수에 저장 프로시듀어 또는 클라이언트 코드의 값이 지정되면 값이 전달되도록 표시기 변수를 0이나 양의 값으로 설정해야 합니다. 예를 들어, 한 출력 전용 매개변수를 취하며 다음과 같이 호출되는 저장 프로시듀어를 가정해봅시다.

```
empind = -1;
EXEC SQL CALL storproc(:empno:empind);
```

저장 프로시듀어에서 첫번째 SQLVAR의 값을 설정하면 결과가 empno로 다시 전달되도록 표시기의 값도 음이 아닌 값으로 설정해야 합니다.

데이터 구조 사용의 요약

표53에서는 저장 프로시듀어 응용프로그램에 따른 다양한 구조 필드의 사용을 요약합니다. 테이블에서 sqlda는 저장 프로시듀어로 전달되는 SQLDA 구조이고 n은 SQLDA의 특정 SQLVAR 구성요소를 지시하는 숫자 값입니다. 오른쪽의 숫자는 테이블 뒤에 나오는 주를 의미합니다.

표 53. 저장 프로시듀어 매개변수

입/출력 SQLDA	sqlda.SQLDAID				4					
	sqlda.SQLDABC				4					
	sqlda.SQLN		2		4					
	sqlda.SQLD		2	3		5				
입/출력 SQLVAR	sqlda.n.SQLTYPE		2	3		5				
	sqlda.n.SQLLEN		2	3		5				
	sqlda.n.SQLDATA	1	2	3			6		8	
	sqlda.n.SQLIND	1	2	3			6		8	9
	sqlda.n.SQLNAME.length						6	7		
	sqlda.n.SQLNAME.data						6	7		
	sqlda.n.SQLDATATYPE_NAME		2	3		5				
	sqlda.n.SQLLONGLEN		2	3		5				
	sqlda.n.SQLDATALEN	1	2	3			6	7		
SQLCA(모든 요소)							6	7		

표 53. 저장 프로시듀어 매개변수 (계속)

주:

저장 프로시듀어를 호출하기 전에 클라이언트 응용프로그램에서는 다음을 수행해야 합니다.

1. SQLTYPE 및 SQLLEN에 기초하여 포인터 구성요소의 저장영역을 할당합니다.
2. 해당 데이터로 구성요소를 초기화합니다.

응용프로그램에서 호출하면 데이터베이스 관리 프로그램에서 다음을 실행합니다.

3. 원래 구성요소의 데이터를 저장 프로시듀어에 할당된 중복 구성요소로 전송합니다. SQLN 구성요소는 SQLD 구성요소의 데이터로 초기화됩니다.

호출시 저장 프로시듀어에서는 다음을 실행할 수 있습니다.

4. 중복 구성요소의 데이터를 변경합니다. 데이터가 유효한 지 점검하거나 클라이언트 응용프로그램으로 데이터를 리턴하지 않으므로 필요에 따라 데이터를 변경할 수 있습니다.

저장 프로시듀어 종료시 데이터베이스 관리 프로그램에서 다음을 실행합니다.

5. 중복 구성요소의 데이터를 점검합니다. 이런 필드의 값이 원래 구성요소의 데이터와 일치하지 않을 경우 오류가 리턴됩니다.
6. 중복 구성요소의 데이터를 원래 구성요소로 리턴합니다.
7. 데이터가 유효한 지 점검하지 않으므로 필요에 따라 데이터를 변경할 수 있습니다.
8. 데이터가 유효한 지 점검하지는 않지만 클라이언트 응용프로그램으로 리턴하므로 필요에 따라 구성요소로 지시된 데이터를 변경할 수 있습니다.
9. SQLTYPE에서 컬럼 유형이 널(NULL) 입력 불가능하다고 지시할 경우 SQLIND 필드는 내부나 외부로 전달되지 않습니다.

입력/출력 SQLDA 및 SQLCA 구조

이 저장 프로시듀어는 SQLDA 구조의 입력 변수로 전달된 정보를 사용하여 수행됩니다. 정보는 SQLDA의 출력 변수로 클라이언트에 리턴됩니다. SQLDA의 SQLD, SQLTYPE 및 SQLLEN 필드의 값은 데이터를 리턴하기 전에 클라이언트 응용프로그램에서 설정한 원래 값과 비교되므로 이런 필드의 값은 변경하지 마십시오. 두 값이 다를 경우 다음 SQLCODE 중 하나가 리턴됩니다.

SQLCODE -1113 (SQLSTATE 39502)

변수의 데이터 유형 즉, SQLTYPE의 값이 변경되었습니다.

SQLCODE -1114 (SQLSTATE 39502)

변수의 길이 즉, SQLLEN의 값이 변경되었습니다.

SQLCODE -1115 (SQLSTATE 39502)

SQLD 필드가 변경되었습니다.

또한 SQLDATA 및 SQLIND 필드에서 지시하는 값을 변경할 수 있더라도 이런 필드의 포인터를 변경하지 마십시오.

주: 입력과 출력에 동일 변수를 사용할 수 있습니다.

저장 프로시듀어에서 리턴하기 전에 저장 프로시듀어의 SQLCA 매개변수로 SQLCA 정보를 명시적으로 복사해야 합니다.

DB2DARI 저장 프로시듀어에 대한 저장 값

저장 프로시듀어의 리턴 값은 클라이언트 응용프로그램으로 리턴되지 않습니다. 나갈 때 서버 프로시듀어를 메모리에서 릴리스해야 할 것인지를 결정할 때 데이터베이스 관리 프로그램에 의해 사용됩니다.

저장 프로시듀어는 다음 값 중 하나를 리턴합니다.

SQLZ_DISCONNECT_PROC
: 데이터베이스 관리 프로그램에게 라이브러리를 릴리스(로드 해제)하도록 지시합니다.

SQLZ_HOLD_PROC
: 데이터베이스 관리 프로그램이 라이브러리가 저장 프로시듀어의 다음 호출 준비가 될 수 있도록 주 메모리에 서버 라이브러리를 보유하도록 지시합니다. 이 명령은 성능을 향상시킬 수 있습니다.

저장 프로시듀어가 한 번만 호출될 경우 SQLZ_DISCONNECT_PROC를 리턴해야 합니다.

클라이언트 응용프로그램이 여러 번의 호출을 발행하여 동일한 저장 프로시듀어를 호출할 경우, SQLZ_HOLD_PROC는 저장 프로시듀어의 리턴 값이 되어야 합니다. 저장 프로시듀어는 로드 해제되지 않습니다.

SQLZ_HOLD_PROC가 사용될 경우, 저장 프로시듀어의 최종 호출은 SQLZ_DISCONNECT_PROC 값을 리턴하여 주 메모리로부터 저장 프로시듀어 라이브러리를 제거해야 합니다. 그렇지 않으면, 라이브러리는 데이터베이스 관리 프로그

램이 중지될 때까지 주 메모리에 남아 있습니다. 저장 프로시듀어에 대한 경고로서 클라이언트 응용프로그램은 최종 호출을 표시하는 매개변수 중 하나로 플래그를 전달할 수 있습니다.

DB2GENERAL UDF 및 저장 프로시듀어

이제부터 PARAMETER STYLE DB2GENERAL UDF 및 Java로 작성된 저장 프로시듀어는 단순히 Java UDF 및 저장 프로시듀어로 언급합니다. DB2GENERAL UDF 및 저장 프로시듀어 작성 프로세스는 지원되는 다른 프로그래밍 언어로 UDF 및 저장 프로시듀어를 작성할 때와 아주 비슷합니다. 일단 작성하여 등록하면 어떤 언어의 프로그램에서든 이를 호출할 수 있습니다. 보통 저장 프로시듀어에서 JDBC API를 호출할 수 있지만 UDF에서는 호출할 수 없습니다.

지원되는 SQL 데이터 유형

PARAMETER STYLE DB2GENERAL UDF 및 저장 프로시듀어를 호출하면 표54에 설명된 대로 DB2에서 자동으로 SQL 유형을 Java 유형으로 또는 그 반대로 변환합니다. 이런 클래스 중 몇 가지가 Java 패키지 `COM.ibm.db2.app`에 나와 있습니다.

표 54. DB2 SQL 유형 및 Java 오브젝트

SQL 컬럼 유형	Java 유형(UDF)	Java 유형(저장 프로시듀어)
SMALLINT (500/501)	short	short
INTEGER (496/497)	int	int
BIGINT (492/493)	long	long
FLOAT (480/481)	double	double
REAL (480/481)[1]	float	float
DECIMAL(p,s) (484/485)	java.math.BigDecimal	java.math.BigDecimal
NUMERIC(p,s) (504/505)	java.math.BigDecimal	java.math.BigDecimal
CHAR(*n*) (452/453)	String	String
CHAR(*n*) FOR BIT DATA (452/453)	Blob	Blob
C null-terminated string (400/401)[2]	n/a	String
VARCHAR(*n*)(448/449)	String	String
VARCHAR(*n*) FOR BIT DATA (448/449)	Blob	Blob
LONG VARCHAR (456/457)	String	String

표 54. DB2 SQL 유형 및 Java 오브젝트 (계속)

SQL 컬럼 유형	Java 유형(UDF)	Java 유형(저장 프로시듀어)
LONG VARCHAR FOR BIT DATA (456/457)	Blob	Blob
GRAPHIC(*n*) (468/469)	String	String
C null-terminated graphic string (460/461)[2]	n/a	String
VARGRAPHIC(*n*) (464/465)	String	String
LONG VARGRAPHIC (472/473)[3]	String	String
BLOB(*n*)(404/405)[3]	Blob	Blob
CLOB(*n*) (408/409)[3]	Clob	Clob
DBCLOB(*n*) (412/413)[3]	Clob	Clob
DATE (384/385)[4]	String	String
TIME (388/389)[4]	String	String
TIMESTAMP (392/393)[4]	String	String

주:

1. SQLDA에서 REAL과 DOUBLE 사이의 차이점은 길이 값(4 또는 8)입니다.
2. C 널 종료 그래픽 문자열과 같이 괄호로 닫힌 유형은 호출 응용프로그램에서 몇 가지 호스트 변수 유형으로 Embedded SQL를 사용할 때 저장 프로시듀어에 나타납니다.
3. BLOB 및 CLOB 클래스는 `COM.ibm.db2.app` 패키지에 나와 있습니다. 그 인터페이스에는 BLOB를 읽고 여기에 쓰는 데 사용할 또는 CLOB용 Reader 및 Writer에 사용할 InputStream 및 OutputStream을 생성하는 루틴이 있습니다. 클래스에 대해서는 865 페이지의 『저장 프로시듀어 및 UDF용 클래스』에서 자세한 내용을 참조하십시오.
4. C로 코딩한 UDF의 경우와 마찬가지로 SQL DATE, TIME 및 TIMESTAMP 값은 Java의 ISO 문자열 엔코딩을 사용합니다.

`COM.ibm.db2.app.Blob` 및 `COM.ibm.db2.app.Clob` 클래스의 인스턴스는 LOB 데이터 유형 즉, BLOB, CLOB 및 DBCLOB를 표시합니다. 이런 클래스에서는 입력으로 전달된 LOB를 읽고 출력으로 리턴된 LOB를 쓰는 제한된 인터페이스를 제공합니다. Java 입출력 스트림 오브젝트를 통해 LOB를 읽고 씁니다. BLOB 클래스의 경우 `getInputStream()` 및 `getOutputStream()` 루틴은 InputStream 또는 OutputStream 오브젝트를 리턴합니다. BLOB 내용은 이 오브젝트를 통해 특정 시기에 바이트 단위로 처리될 수 있습니다. CLOB의 경우 `getReader()` 및

getWriter() 루틴은 Reader 또는 Writer 오브젝트를 리턴합니다. CLOB 또는 DBCLOB 내용은 이 오브젝트를 통해 특정 시기에 문자 단위로 처리될 수 있습니다.

set() 메소드를 사용하여 이런 오브젝트를 출력으로 리턴할 경우 데이터베이스 코드 페이지의 Java 유니코드 문자를 표시하도록 코드 페이지 변환이 적용될 수 있습니다.

저장 프로시듀어 및 UDF용 클래스

Java 저장 프로시듀어는 Java UDF와 아주 비슷합니다. 테이블 함수와 같이 이 저장 프로시듀어의 출력은 여러 가지일 수 있습니다. 또한 널(NULL) 값에 대해 동일 변환을 사용하고 출력을 위해 동일 set 루틴을 사용합니다. 주요 차이점은 저장 프로시듀어를 포함한 Java 클래스는 COM.ibm.db2.app.UDF 클래스 대신 COM.ibm.db2.app.StoredProc 클래스에서 계승해야 합니다. COM.ibm.db2.app.StoredProc 클래스에 대해서는 866 페이지의 『COM.ibm.db2.app.StoredProc』에서 자세한 내용을 참조하십시오.

이 인터페이스에서는 Embedding 응용프로그램 문맥에 대한 JDBC 연결을 가져오는 다음 루틴을 제공합니다.

```
public java.sql.Connection getConnection()
```

이 핸들을 사용하여 SQL문을 수행할 수 있습니다. StoredProc 인터페이스의 다른 메소드는 sqllib/samples/java/StoredProc.java 파일에 나열됩니다.

Java 저장 프로시듀어 또는 UDF에 사용할 수 있는 클래스/인터페이스는 다음과 같이 5가지입니다.

- COM.ibm.db2.app.StoredProc
- COM.ibm.db2.app.UDF
- COM.ibm.db2.app.Lob
- COM.ibm.db2.app.Blob
- COM.ibm.db2.app.Clob

다음 절에서는 이런 클래스 수행의 공용 측면에 대해 설명합니다.

COM.ibm.db2.app.StoredProc

PARAMETER STYLE DB2GENERAL 저장 프로시듀어로 호출되는 메소드를 포함한 Java 클래스는 public이어야 하며 이 Java 인터페이스를 구현해야 합니다. 이런 클래스는 다음과 같이 선언해야 합니다.

```
public class <user-STP-class> extends COM.ibm.db2.app.StoredProc{ ... }
```

현재 실행 중인 저장 프로시듀어의 문맥에서 COM.ibm.db2.app.StoredProc 인터페이스의 계승된 메소드만을 호출할 수 있습니다. 예를 들어, 저장 프로시듀어에서 리턴한 후에는 LOB 인수, 결과 또는 상태 설정 호출 등에 대해 작업할 수 없습니다. 이 규칙을 위반할 경우 Java 예외가 발생합니다.

인수 관련 호출에서는 컬럼 색인을 사용하여 참조 중인 컬럼을 식별합니다. 컬럼 색인은 첫번째 인수에 대해 1에서 시작합니다. PARAMETER STYLE DB2GENERAL 저장 프로시듀어의 모든 인수는 INOUT으로 간주되므로 입력이자 출력입니다.

저장 프로시듀어에서 리턴된 예외는 데이터베이스에서 포착하며 SQLCODE -4302, SQLSTATE 38501과 함께 호출자에 리턴됩니다. JDBC SQLException 또는 SQLWarning은 특별하게 처리되며 자체 SQLCODE, SQLSTATE 등을 호출 응용프로그램 verbatim으로 전달합니다.

다음 메소드는 COM.ibm.db2.app.StoredProc 클래스와 연관되어 있습니다.

```
public StoredProc() [default constructor]
```

저장 프로시듀어 호출 전에 데이터베이스에서 이 생성자를 호출합니다.

```
public boolean isNull(int) throws Exception
```

이 함수는 지정된 색인이 포함된 입력 인수가 SQL 널(NULL)인지 테스트합니다.

```
public void set(int, short) throws Exception
public void set(int, int) throws Exception
public void set(int, double) throws Exception
public void set(int, float) throws Exception
public void set(int, java.math.BigDecimal) throws Exception
public void set(int, String) throws Exception
public void set(int, COM.ibm.db2.app.Blob) throws Exception
public void set(int, COM.ibm.db2.app.Clob) throws Exception
```

이 함수는 지정된 색인이 포함된 출력 인수를 지정된 값으로 설정합니다. 색인은 유효한 출력 인수를 참조해야 하며, 데이터 유형이 일치해야 하고, 값의 길이와 내용은 수용 가능해야 합니다. 유니코드 문자가 포함된 문자열은 데이터베이스 코드 페이지에서 표시 가능해야 합니다. 오류는 예외 발생을 낳습니다.

```
public java.sql.Connection getConnection() throws Exception
```

이 함수는 데이터베이스에 대한 호출 응용프로그램의 연결을 표시하는 JDBC 오브젝트를 리턴합니다. 이는 C 저장 프로시듀어에서 널(NULL) `SQLConnect()`를 호출한 결과와 비슷합니다.

COM.ibm.db2.app.UDF

PARAMETER STYLE DB2GENERAL UDF로 호출되는 메소드를 포함한 Java 클래스는 public이어야 하며 이 Java 인터페이스를 구현해야 합니다. 이런 클래스는 다음과 같이 선언해야 합니다.

```
public class <user-UDF-class> extends COM.ibm.db2.app.UDF{ ... }
```

현재 실행 중인 UDF의 문맥에서 `COM.ibm.db2.app.UDF` 인터페이스의 메소드만을 호출할 수 있습니다. 예를 들어, UDF에서 리턴한 후에는 LOB 인수, 결과 또는 상태 설정 호출 등에 대해 작업할 수 없습니다. 이 규칙을 위반할 경우 Java 예외가 발생합니다.

인수 관련 호출에서는 컬럼 색인을 사용하여 설정 중인 컬럼을 식별합니다. 컬럼 색인은 첫번째 인수에 대해 1에서 시작합니다. 출력 인수에는 입력 인수보다 높은 번호가 지정됩니다. 예를 들어, 세 개의 입력이 있는 스칼라 UDF는 출력에 색인 4를 사용합니다.

UDF에서 리턴된 예외는 데이터베이스에서 포착하며 SQLCODE -4302, SQLSTATE 38501과 함께 호출자에 리턴됩니다.

다음 메소드는 `COM.ibm.db2.app.UDF` 클래스와 연관되어 있습니다.

```
public UDF() [default constructor]
```

일련의 UDF 호출 시작시 데이터베이스에서 이 생성자를 호출합니다. 이는 첫번째 UDF 호출에 앞섭니다.

```
public void close()
```

FINAL CALL 옵션으로 UDF를 작성할 때 UDF 평가 종료시 데이터베이스에서 이 함수를 호출합니다. 이는 C UDF를 마지막으로 호출한 결과와 비슷합니다. 테이블 함수의 경우 UDF 메소드에 대한 CLOSE 호출 후(NO FINAL CALL이 코딩되거나 기본값으로 설정될 경우) 또는 FINAL 호출 후 close()가 호출됩니다. Java UDF 클래스에서 이 함수를 구현하지 못할 경우 no-op 스텁에서 이 이벤트를 조절하여 무시합니다.

```
public int getCallType() throws Exception
```

테이블 함수 UDF에서는 getCallType()를 사용하여 특정 호출에 대해 호출 유형을 찾습니다. 그리고 다음과 같이 값을 리턴합니다(COM.ibm.db2.app.UDF 클래스 정의에 있는 해당 값에 대해 기호 정의가 제공됩니다).

- -2 FIRST call
- -1 OPEN call
- 0 FETCH call
- 1 CLOSE call
- 2 FINAL call

```
public boolean isNull(int) throws Exception
```

이 함수는 지정된 색인이 포함된 입력 인수가 SQL 널(NULL)인지 테스트합니다.

```
public boolean needToSet(int) throws Exception
```

이 함수는 지정된 색인이 포함된 출력 인수를 설정해야 하는지 여부를 테스트합니다. UDF 호출자에서 그 컬럼을 사용하지 않을 경우 DBINFO로 선언된 테이블 UDF에 대해 false가 될 수 있습니다.

```
public void set(int, short) throws Exception
public void set(int, int) throws Exception
public void set(int, double) throws Exception
public void set(int, float) throws Exception
public void set(int, java.math.BigDecimal) throws Exception
public void set(int, String) throws Exception
public void set(int, COM.ibm.db2.app.Blob) throws Exception
public void set(int, COM.ibm.db2.app.Clob) throws Exception
```

이 함수는 지정된 색인이 포함된 출력 인수를 지정된 값으로 설정합니다. 색인은 유효한 출력 인수를 참조해야 하며, 데이터 유형이 일치해야 하고, 값의 길이와 내

용은 수용 가능해야 합니다. 유니코드 문자가 포함된 문자열은 데이터베이스 코드 페이지에서 표시 가능해야 합니다. 오류는 예외 발생을 낳습니다.

```
public void setSQLstate(String) throws Exception
```

이 함수가 UDF에서 호출되어, 이 호출에서 리턴되도록 SQLSTATE를 설정할 수 있습니다. 테이블 UDF에서 "02000"으로 이 함수를 호출하여 테이블 끝 조건을 신호로 보내야 합니다. 문자열이 SQLSTATE로 수용 가능하지 않을 경우 예외가 발생합니다.

```
public void setSQLmessage(String) throws Exception
```

이 함수는 setSQLstate 함수와 비슷합니다. 이 함수는 SQL 메시지 결과를 설정합니다. 문자열이 70자보다 길 경우와 같이 수용 가능하지 않을 경우 예외가 발생합니다.

```
public String getFunctionName() throws Exception
```

이 함수는 실행 중인 UDF의 이름을 리턴합니다.

```
public String getSpecificName() throws Exception
```

이 함수는 실행 중인 UDF의 특정 이름을 리턴합니다.

```
public byte[] getDBinfo() throws Exception
```

이 함수는 실행 중인 UDF의 처리되지 않은 DBINFO 구조를 바이트 배열로 리턴합니다. 먼저 DBINFO 옵션으로 이 함수를 선언해야 합니다.

```
public String getDBname() throws Exception
public String getDBauthid() throws Exception
public String getDBtbschema() throws Exception
public String getDBtbname() throws Exception
public String getDBcolname() throws Exception
public String getDBver_rel() throws Exception
public String getDBplatform() throws Exception
public String getDBapplid() throws Exception
```

이런 함수는 실행 중인 UDF의 DBINFO 구조에서 해당 필드의 값을 리턴합니다.

```
public int[] getDBcodepg() throws Exception
```

이 함수는 DBINFO 구조에서 SBCS, DBCS 및 데이터베이스의 복합 코드 페이지 수를 리턴합니다. 리턴된 정수 배열에는 처음 세 개의 구성요소로 각 숫자가 있습니다.

```
public byte[] getScratchpad() throws Exception
```

이 함수는 현재 실행 중인 UDF의 스크래치 패드 사본을 리턴합니다. 먼저 SCRATCHPAD 옵션으로 UDF를 선언해야 합니다.

```
public void setScratchpad(byte[]) throws Exception
```

이 함수는 지정된 바이트 배열의 내용으로 현재 실행 중인 UDF의 스크래치 패드를 겹쳐씁니다. 먼저 SCRATCHPAD 옵션으로 UDF를 선언해야 합니다. 바이트 배열의 크기는 `getScratchpad()` 리턴과 동일합니다.

COM.ibm.db2.app.Lob

이 클래스는 사용자 정의 함수 또는 저장 프로시듀어에서의 계산을 위해 임시 BLOB 또는 CLOB 오브젝트를 작성하는 유틸리티 루틴을 제공합니다.

다음 메소드는 COM.ibm.db2.app.Lob 클래스와 연관됩니다.

```
public static Blob newBlob() throws Exception
```

이 함수는 임시 BLOB를 작성합니다. 가능한 경우 LOCATOR 을 사용하여 이 함수를 구현합니다.

```
public static Clob newClob() throws Exception
```

이 함수는 임시 CLOB을 작성합니다. 가능한 경우 LOCATOR 을 사용하여 이 함수를 구현합니다.

COM.ibm.db2.app.Blob

UDF 또는 저장 프로시듀어 입력으로 BLOB를 표시하도록 데이터베이스에서 이 클래스의 인스턴스를 전달하며 출력으로 전달할 수 있습니다. 응용프로그램에서 인스턴스를 작성할 수 있지만 실행 중인 UDF 또는 저장 프로시듀어의 문맥에서만 그렇게 할 수 있습니다. 이런 문맥 외부에서 해당 오브젝트를 사용하면 예외가 발생합니다.

다음 메소드는 `COM.ibm.db2.app.Blob` 클래스와 연관됩니다.

```
public long size() throws Exception
```

이 함수는 BLOB의 길이를 바이트 단위로 리턴합니다.

```
public java.io.InputStream getInputStream() throws Exception
```

이 함수를 새로운 InputStream을 리턴하여 BLOB의 문맥을 읽습니다. 효율적인 seek/mark 작업은 오브젝트에서 사용 가능합니다.

```
public java.io.OutputStream getOutputStream() throws Exception
```

이 함수는 새로운 OutputStream을 리턴하여 BLOB에 바이트를 첨부합니다. 첨부된 바이트는 즉시 이 오브젝트의 `getInputStream()` 호출로 생성된 기존의 모든 InputStream 인스턴스에 가시적이 됩니다.

COM.ibm.db2.app.Clob

UDF 또는 저장 프로시듀어 입력으로 CLOB를 표시하도록 데이터베이스에서 이 클래스의 인스턴스를 전달하며 출력으로 전달할 수 있습니다. 응용프로그램에서 인스턴스를 작성할 수 있지만 실행 중인 UDF 또는 저장 프로시듀어의 문맥에서만 그렇게 할 수 있습니다. 이런 문맥 외부에서 해당 오브젝트를 사용하면 예외가 발생합니다.

CLOB 인스턴스는 데이터베이스 코드 페이지에 문자를 저장합니다. 일부 유니코드 문자가 이 코드 페이지에서 표시되지 않을 수 있으며 변환 중 예외를 발생시킬 수도 있습니다. 첨부 작업 중이나 UDF 또는 StoredProc `set()` 호출 중 이런 경우가 발생할 수 있습니다. 이런 경우 Java 프로그래머가 CLOB와 DBCLOB 사이의 차이점은 숨겨야 합니다.

다음 메소드는 `COM.ibm.db2.app.Clob` 클래스와 연관됩니다.

```
public long size() throws Exception
```

이 함수는 CLOB의 길이를 문자 단위로 리턴합니다.

```
public java.io.Reader getReader() throws Exception
```

이 함수는 새로운 Reader를 리턴하여 CLOB 또는 DBCLOB의 문맥을 읽습니다. 효율적인 seek/mark 작업은 오브젝트에서 사용 가능합니다.

```
public java.io.Writer getWriter() throws Exception
```

이 함수는 새로운 Writer를 리턴하여 이 CLOB 또는 DBCLOB에 이 문자를 첨부합니다. 첨부된 문자는 즉시 이 오브젝트의 `GetReader()` 호출로 생성된 기존의 모든 Reader 인스턴스에 가시적이 됩니다.

비분리 저장 프로시듀어

DB2DARI 저장 프로시듀어를 비분리 저장 프로시듀어로 수행해야 한다는 사실을 지시하려면 *응용프로그램 빌드 안내서*에 지시된 디렉토리에 넣으십시오. 비분리 저장 프로시듀어에 대해서는 249 페이지의 『NOT FENCED 저장 프로시듀어』에서 자세한 내용을 참조하십시오.

예제 입력-SQLDA 프로그램

다음은 입력 SQLDA 구조 사용의 데모를 나타내는 샘플 프로그램입니다. 클라이언트 응용프로그램은 `Presidents` 테이블을 작성하고 데이터와 함께 테이블을 로드하는 저장 프로시듀어를 호출합니다.

이 프로그램은 SAMPLE 데이터베이스의 `Presidents` 테이블을 작성합니다. 그런 다음 테이블에 `Washington`, `Jefferson` 및 `Lincoln` 값을 삽입합니다.

저장 프로시듀어를 사용하지 않을 경우 샘플 프로그램은 그림23과 같이 각 SQL 문을 처리하기 위해 네트워크에서 네 개의 별도 요청으로 데이터를 전송하도록 설계되었습니다.

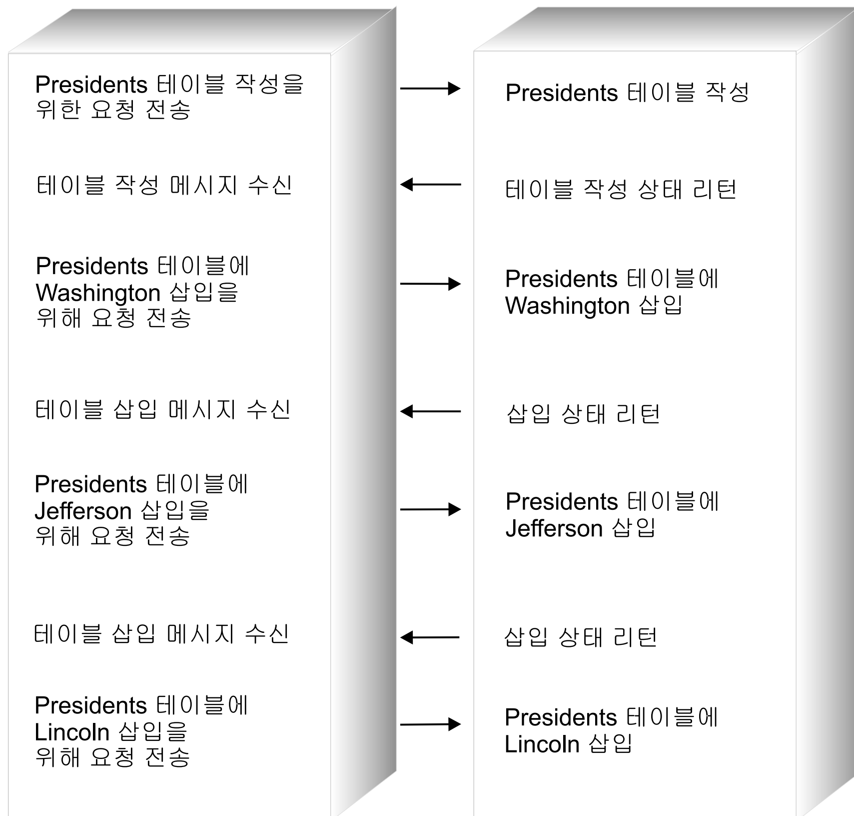


그림 23. 저장 프로시듀어가 없는 입력-SQLDA 샘플

대신 샘플 프로그램에서는 저장 프로시듀어 기술을 활용하여 서버 프로시듀어가 SQL문을 한 그룹으로 실행하도록 허용하면서 네트워크에서 모든 데이터를 한 요청으로 전송합니다. 이 기술은 그림24에 나와 있습니다.

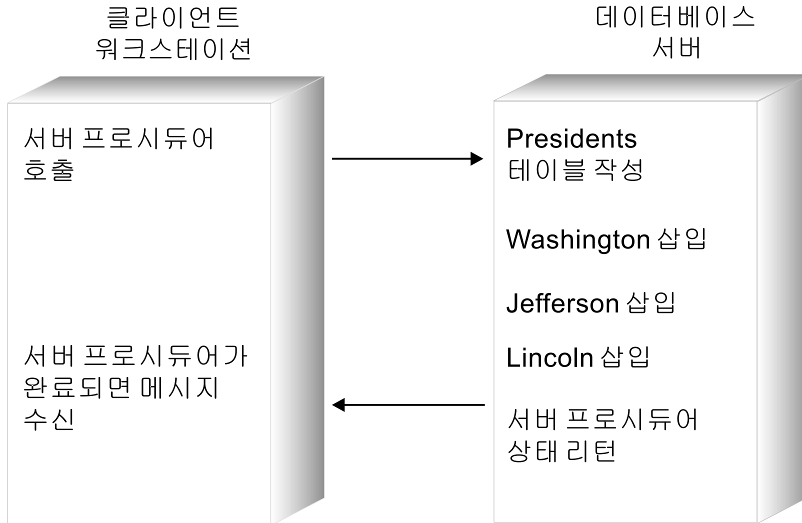


그림 24. 저장 프로시듀어가 있는 입력-SQLDA 샘플

샘플 입력-SQLDA 클라이언트 응용프로그램 및 샘플 입력-SQLDA 저장 프로시듀어는 874에 나와 있습니다.

입력-SQLDA 클라이언트 응용프로그램의 작업 방식

1. **입력 SQLDA 구조 초기화**. 다음과 같은 입력 SQLDA의 필드가 초기화됩니다.
   - SQLN 및 SQLD 구성요소는 할당되고 사용된 SQLVAR 구성요소의 총수로 설정됩니다.
   - SQLTYPE 구성요소는 문자 데이터 유형을 지시하도록 설정됩니다.
   - 첫번째 SQLDATA 구성요소는 테이블의 이름으로 설정됩니다. 두 번째에서 네 번째 SQLDATA 구성요소는 `Washington`, `Jefferson` 및 `Lincoln` 값으로 설정됩니다.
   - SQLLEN 구성요소는 각 SQLDATA 구성요소의 길이로 설정됩니다(C 언어 널(NULL) 종료기가 있을 경우 1 바이트 추가).
   - SQLIND 구성요소는 `NULL`로 설정됩니다.

2. **서버 프로시듀어 호출**. 응용프로그램에서 다음을 사용하여 sample 데이터베이스 위치에 있는 inpsrv 프로시듀어를 호출합니다.
   a. 호스트 변수가 있는 CALL문
   b. SQLDA가 있는 CALL문

CHECKERR 매크로/함수는 오류 체크 유틸리티입니다. 이 오류 체크 유틸리티의 위치는 사용된 프로그래밍 언어에 따라 달라질 수 있습니다.

C DB2 API를 호출하는 C 프로그램에 대해서는 utilapi.c에서 sqlInfoPrint 함수는 utilapi.h에서 API_SQL_CHECK로서 참조됩니다. C embedded SQL 프로그램에 대해서는 utilemb.sqc에서 sqlInfoPrint 함수는 utilemb.h에서 EMB_SQL_CHECK로서 참조됩니다.

COBOL CHECKERR은 checkerr.cbl라고 하는 외부 프로그램입니다.

FORTRAN CHECKERR은 util.f 파일에 있는 서브루틴입니다.

REXX CHECKERR은 현재 프로그램의 끝에 있습니다.

이 오류 체크 유틸리티에 대한 소스 코드에 대해서는 137 페이지의 『예제 프로그램에서 GET ERROR MESSAGE 사용』에서 자세한 내용을 참조하십시오.

C 예: V5SPCLI.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlca.h>
#include <sqlda.h>
#include <sqlutil.h>
#include "util.h"

#define  CHECKERR(CE_STR)   if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION;
     char database[9];
     char userid[9];
     char passwd[19];
     char procname[255]  = "inpsrv";
     char table_name[11]    = "PRESIDENTS";
     char data_item0[21]    = "Washington";
     char data_item1[21]    = "Jefferson";
     char data_item2[21]    = "Lincoln";
     short tableind, dataind0, dataind1, dataind2;
   EXEC SQL END DECLARE SECTION;

   /* Declare Variables for CALL USING */
   struct sqlca    sqlca;
   struct sqlda   *inout_sqlda = NULL;
   char eBuffer[1024]; /* error message buffer */


   if (argc != 4) {
      printf ("\nUSAGE: inpcli remote_database userid passwd\n\n");
      return 1;
   }

   strcpy (database, argv[1]);
   strcpy (userid, argv[2]);
   strcpy (passwd, argv[3]);
   /* Connect to Remote Database */
   printf("CONNECT TO Remote Database.\n");
   EXEC SQL CONNECT TO :database USER :userid USING :passwd;
   CHECKERR ("CONNECT TO SAMPLE");

   /*****************************************************\
   * Call the Remote Procedure via CALL with Host Variables *
   \*****************************************************/
   printf("Use CALL with Host Variable to invoke the Server Procedure"
      " named inpsrv.\n");
   tableind = dataind0 = dataind1 = dataind2 = 0;

   EXEC SQL CALL :procname (:table_name:tableind, :data_item0:dataind0,
      :data_item1:dataind1, :data_item2:dataind2);  2a
   /* COMMIT or ROLLBACK the transaction */
   if (SQLCODE == 0)
   { EXEC SQL COMMIT;
     printf("Server Procedure Complete.\n\n");
```

```
}
else
{ /* print the error message, roll back the transaction and return */
  sqlaintp (eBuffer, 1024, 80, &sqlca);
  printf("\n%s\n", eBuffer);

  EXEC SQL ROLLBACK;
  printf("Server Procedure Transaction Rolled Back.\n\n");
  return 1;
}


/* Allocate and Initialize Input SQLDA */    1
inout_sqlda = (struct sqlda *)malloc( SQLDASIZE(4) );
inout_sqlda->sqln = 4;
inout_sqlda->sqld = 4;

inout_sqlda->sqlvar[0].sqltype = SQL_TYP_NCSTR;
inout_sqlda->sqlvar[0].sqldata = table_name;
inout_sqlda->sqlvar[0].sqllen  = strlen( table_name ) + 1;
inout_sqlda->sqlvar[0].sqlind  = &tableind;

inout_sqlda->sqlvar[1].sqltype = SQL_TYP_NCSTR;
inout_sqlda->sqlvar[1].sqldata = data_item0;
inout_sqlda->sqlvar[1].sqllen  = strlen( data_item0 ) + 1;
inout_sqlda->sqlvar[1].sqlind  = &dataind0;

inout_sqlda->sqlvar[2].sqltype = SQL_TYP_NCSTR;
inout_sqlda->sqlvar[2].sqldata = data_item1;
inout_sqlda->sqlvar[2].sqllen  = strlen( data_item1 ) + 1;
inout_sqlda->sqlvar[2].sqlind  = &dataind1;

inout_sqlda->sqlvar[3].sqltype = SQL_TYP_NCSTR;
inout_sqlda->sqlvar[3].sqldata = data_item2;
inout_sqlda->sqlvar[3].sqllen  = strlen( data_item2 ) + 1;
inout_sqlda->sqlvar[3].sqlind  = &dataind2;

/**********************************************\
* Call the Remote Procedure via CALL with SQLDA *
\**********************************************/
printf("Use CALL with SQLDA to invoke the Server Procedure named "
   "inpsrv.\n");

tableind = dataind0 = dataind1 = dataind2 = 0;
inout_sqlda->sqlvar[0].sqlind  = &tableind;
inout_sqlda->sqlvar[1].sqlind  = &dataind0;
inout_sqlda->sqlvar[2].sqlind  = &dataind1;
inout_sqlda->sqlvar[3].sqlind  = &dataind2;

EXEC SQL CALL :procname USING DESCRIPTOR :*inout_sqlda;   2b
/* COMMIT or ROLLBACK the transaction */
if (SQLCODE == 0)
{ EXEC SQL COMMIT;
  printf("Server Procedure Complete.\n\n");
}
else
{ /* print the error message, roll back the transaction and return */
  sqlaintp (eBuffer, 1024, 80, &sqlca);
  printf("\n%s\n", eBuffer);
```

```
    EXEC SQL ROLLBACK;
    printf("Server Procedure Transaction Rolled Back.\n\n");
    return 1;
  }

  /* Free allocated memory */
  free( inout_sqlda );

  /* Drop the PRESIDENTS table created by the stored procedure */
  EXEC SQL DROP TABLE PRESIDENTS;
  CHECKERR("DROP TABLE");

  /* Disconnect from Remote Database */
  EXEC SQL CONNECT RESET;
  CHECKERR ("CONNECT RESET");
  return 0;
}
/* end of program : inpcli.sqc */
```

예제 입력-SQLDA 저장 프로시듀어의 작업 방식

1. **서버 프로시듀어 선언**. 프로시듀어는 SQLDA 및 SQLCA 구조의 포인터를 수용합니다.
2. **테이블 작성**. SQLDA 구조의 첫번째 SQLVAR에 전달된 데이터를 사용하여 CREATE TABLE문이 생성되고 Presidents 테이블을 작성하도록 실행됩니다.
3. **삽입 명령문 준비**. 매개변수 표시문자 ?가 표시된 INSERT문이 준비됩니다.
4. **데이터 삽입**. SQLDA 구조의 두 번째에서 네 번째 SQLVAR에 전달된 데이터를 사용하여 이전에 준비된 INSERT문을 실행합니다. 매개변수 표시문자는 Washington, Jefferson 및 Lincoln 값으로 대체됩니다. 이런 값은 Presidents 테이블에 삽입됩니다.
5. **클라이언트 응용프로그램으로 리턴**. 서버 프로시듀어에서 클라이언트 응용프로그램의 SQLCA로 SQLCA를 복사하고, 트랜잭션이 성공한 경우 COMMIT문을 발행하고, 서버 프로시듀어에 대한 더 이상의 호출이 없음을 지시하는 SQLZ_DISCONNECT_PROC 값을 리턴합니다.

주: 서버 프로시듀어는 AIX 시스템에서 REXX로 작성될 수 없습니다.

C 예: V5SPSRV.SQC

```
#include <memory.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlutil.h>

#ifdef __cplusplus
extern "C"
#endif
SQL_API_RC SQL_API_FN inpsrv(void *reserved1,          1
           void *reserved2,
           struct sqlda   *inout_sqlda,
           struct sqlca   *ca)
{
   /* Declare a local SQLCA */
   EXEC SQL INCLUDE SQLCA;

   /* Declare Host Variables */
   EXEC SQL BEGIN DECLARE SECTION;
     char table_stmt[80]  = "CREATE TABLE ";
     char insert_stmt[80] = "INSERT INTO ";
     char insert_data[21];
   EXEC SQL END DECLARE SECTION;

   /* Declare Miscellanous Variables */
   int  cntr = 0;
   char *table_name;
   char *data_items[3];
   short  data_items_length[3];
   int  num_of_data = 0;

   /*-------------------------------------------------------------------*/
   /* Assign the data from the SQLDA to local variables so that we    */
   /* don't have to refer to the SQLDA structure further.  This will  */
   /* provide better portability to other platforms such as DB2 MVS   */
   /* where they receive the parameter list differently.              */
   /*-------------------------------------------------------------------*/


   table_name  = inout_sqlda->sqlvar[0].sqldata;
   num_of_data = inout_sqlda->sqld - 1;

   for (cntr = 0; cntr < num_of_data; cntr++)
   {
      data_items[cntr] = inout_sqlda->sqlvar[cntr+1].sqldata;
      data_items_length[cntr] = inout_sqlda->sqlvar[cntr+1].sqllen;
   }

   /*-------------------------------------------------------------------*/
   /* Create President Table                                          */
   /* - For simplicity, we'll ignore any errors from the              */
   /*   CREATE TABLE so that you can run this program even when the   */
   /*   table already exists due to a previous run.                   */
   /*-------------------------------------------------------------------*/
```

```
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   strcat(table_stmt, table_name);
   strcat(table_stmt, " (name CHAR(20))");   2

   EXEC SQL EXECUTE IMMEDIATE :table_stmt;

   EXEC SQL WHENEVER SQLERROR GOTO ext;

   /*-----------------------------------------------------------------*/
   /* Generate and execute a PREPARE for an INSERT statement, and     */
   /* then insert the three presidents.                               */
   /*-----------------------------------------------------------------*/

   strcat(insert_stmt, table_name );
   strcat(insert_stmt, "  VALUES (?)");   3

   EXEC SQL PREPARE S1 FROM :insert_stmt;

   for (cntr = 0; cntr < num_of_data; cntr++)
   {
      strncpy(insert_data, data_items[cntr], data_items_length[cntr]);
      insert_data[data_items_length[cntr]] = '\0';
      EXEC SQL EXECUTE S1 USING :insert_data;   4
   }

   /*-----------------------------------------------------------------*/
   /* Return to caller                                                */
   /*   -  Copy the SQLCA                                             */
   /*   -  Update the output SQLDA.  Since there's no output to       */
   /*      return, we are setting the indicator values to -128 to    */
   /*      return only a null value.                                  */
   /*-----------------------------------------------------------------*/

ext:   5
   memcpy(ca, &sqlca, sizeof(struct sqlca));
   if (inout_sqlda != NULL)
   {
     for (cntr = 0; cntr < inout_sqlda->sqld; cntr++)
     {
       *(inout_sqlda->sqlvar[cntr].sqlind) = -128;
     }
   }

   return(SQLZ_DISCONNECT_PROC);
}
```

부록D. 호스트 또는 AS/400 환경에서 프로그래밍

이 절에는 *DB2 Connect 사용자 안내서*에 공통적인 내용이 있습니다. 이 절에 익숙하지 않은 용어나 개념이 있으면 *DB2 Connect 사용자 안내서*에서 자세한 내용을 참조하십시오.

DB2 Connect를 통해 응용프로그램은 System/390 및 AS/400 서버에 있는 DB2 데이터베이스에서 데이터를 액세스할 수 있습니다. 예를 들어, Windows에서 실행하는 응용프로그램은 OS/390용 DB2 Universal Database 데이터베이스에 있는 데이터를 액세스할 수 있습니다. 호스트 또는 AS/400 환경에서 수행하도록 새로운 응용프로그램을 작성하거나 기존 응용프로그램을 수정할 수 있습니다. 또한 한 환경에서 응용프로그램을 개발하여 다른 환경으로 이식할 수도 있습니다.

DB2 Connect를 통해 호스트 데이터베이스 제품에서 항목을 지원하는 한 OS/390용 DB2 Universal Database과 같은 호스트 데이터베이스 제품과 함께 다음 API를 사용할 수 있습니다.

- 정적 및 동적 Embedded SQL
- DB2 Call Level Interface
- Microsoft ODBC API
- JDBC.

일부 SQL문은 관계형 데이터베이스 제품에 따라 다릅니다. 다음과 같은 SQL문을 만날 수 있습니다.

- 표준에 관계 없이 사용하는 모든 데이터베이스 제품에 동일합니다.
- *SQL 참조서*에 설명되어 있으므로 모든 OBM 관계형 데이터베이스 제품에서 사용할 수 있습니다.
- 액세스하는 한 데이터베이스 시스템에 고유합니다.

© Copyright IBM Corp. 1993, 2000

처음 두 범주의 SQL문은 쉽게 이식 가능하지만 세 번째 범주의 SQL문의 경우에는 먼저 변경해야 합니다. 일반적으로 DDL(Data Definition Language)의 SQL 문은 DML(Data Manipulation Language)의 SQL문 만큼 쉽게 이식할 수 없습니다.

DB2 Connect는 DB2 Universal Database에서 지원하지 않는 일부 SQL문을 수용합니다. DB2 Connect에서는 이런 명령문으로 호스트나 AS/400 서버로 전달합니다. 최대 컬럼 길이와 같은 여러 플랫폼의 제한사항에 대해서는 *SQL 참조서*에서 자세한 내용을 참조하십시오.

AIX용 CICS와 같은 다른 CICS 제품을 수행하도록 OS/390에서 VSE로 CICS 응용프로그램을 이동할 경우 DB2 Connect를 사용하여 OS/390 또는 VSE 데이터베이스에 액세스할 수도 있습니다. 더 자세한 내용은 *CICS/6000 Application Programming Guide* 및 *CICS Customization and Operation* 지침서를 참조하십시오.

주: DB2 Connect 없이 DB2 개인용 프로토콜을 사용하는 것이 더 효율적이지만 DB2 Universal Database 버전 7 데이터베이스에 DB2 Connect를 사용할 수 있습니다. 추상 데이터 유형에 대한 비지원과 같이 DB2 Connect 자체의 제한사항으로 인해 제한이 발생할 경우를 제외하고 DB2 Universal Database 버전 7 데이터베이스에 대해 DB2 Connect를 사용할 경우 다음 절에 나열되는 대부분의 비호환성 문제는 적용되지 않습니다.

호스트 또는 AS/400 환경에서 프로그래밍할 때 다음과 같은 특정 요인을 고려해야 합니다.

- DDL(Data Definition Language) 사용
- DML(Data Manipulation Language) 사용
- DCL(Data Control Language) 사용
- 연결 및 연결해제
- 사전 처리 컴파일
- 정렬 순서 정의
- 참조 무결성 관리
- 잠금

- SQLCODE와 SQLSTATE 사이의 차이점
- 시스템 카탈로그 사용
- 분리 레벨
- 저장 프로시듀어
- 최소단위 복합 SQL이 아님
- 분산 작업 단위(DUOW)
- DB2 Connect에서 지원하거나 거부하는 SQL문

DDL(Data Definition Language) 사용

시스템에 따라 저장영역이 다르게 처리되므로 DDL문은 IBM 데이터베이스 사이에서 달라집니다. 호스트 또는 AS/400 서버 시스템에서 데이터베이스 지정과 CREATE TABLE문 발행 사이에 몇 개의 단계가 있을 수 있습니다. 예를 들어, 일련의 명령문으로 논리 오브젝트의 디자인을 저장영역의 해당 오브젝트에 대한 물리 표시로 변환할 수 있습니다.

호스트 또는 AS/400 서버 데이터베이스로 사전 처리 컴파일할 때 사전 처리 컴파일러에서 DDL문 같은 여러 명령문을 호스트 또는 AS/400 서버에 전달합니다. 응용프로그램이 수행 중인 데이터베이스에 대해서는 동일 명령문으로도 사전 처리 컴파일하지 못합니다. 예를 들어, OS/2 응용프로그램에서 CREATE STORGROUP 문으로는 OS/2용 DB2 데이터베이스가 아닌 OS/390용 DB2 Universal Database 데이터베이스로 사전 처리 컴파일합니다.

DML(Data Manipulation Language) 사용

일반적으로 DML문의 이식성은 높습니다. SELECT, INSERT, UPDATE 및 DELETE문은 IBM 관계형 데이터베이스 제품에 걸쳐 비슷합니다. 대부분의 응용프로그램은 기본적으로 DB2 Connect 프로그램에서 지원하는 DML SQL문을 사용합니다.

숫자 데이터 유형

DB2 Universal Database로 숫자 데이터가 전송되면 데이터 유형이 변경될 수 있습니다. AS/400용 DB2 Universal Database에서 지원하는 숫자 및 구역화된 십진 SQLTYPE는 고정(팩된) 10진 SQLTYPE로 변환됩니다.

혼합 바이트 데이터

혼합 바이트 데이터는 동일 컬럼에 있는 EUC(extended UNIX code) 문자 세트, 2바이트 문자 세트(DBCS) 및 1바이트 문자 세트(SBCS)의 문자로 구성될 수 있습니다. OS/390, OS/400, VSE 및 VM과 같이 EBCDIC에 데이터를 저장하는 시스템에서 shift-out 및 shift-in 문자는 2 바이트 데이터의 시작과 끝을 표시합니다. OS/2 및 UNIX와 같이 ASCII로 데이터를 저장하는 시스템에서 shift-in 및 shift-out 문자는 필요하지 않습니다.

응용프로그램이 ASCII 시스템에서 EBCDIC 시스템으로 혼합 바이트 데이터를 전송할 경우 shift 문자에 충분한 공간을 허용해야 합니다. SBCS에서 DBCS 데이터로의 각 전환에 대해 데이터 길이에 2 바이트가 추가하십시오. 더 나은 이식성을 위해 혼합 바이트 데이터를 사용하는 응용프로그램에서 가변 길이 문자열을 사용하십시오.

긴 필드

254자보다 긴 문자열의 긴 필드는 시스템에 따라 다르게 처리됩니다. 호스트 또는 AS/400 서버에서 긴 필드에 대해 스칼라 함수의 부속 집합만을 지원할 수 있습니다. 예를 들어, OS/390용 DB2 Universal Database에서는 긴 필드에 대해 **LENGTH** 및 **SUBSTR** 함수만을 허용합니다. 또한 호스트 또는 AS/400 서버에서 특정 SQL문에 대해 다른 처리를 요구할 수 있습니다. 예를 들어, VSE & VM용 DB2의 경우 INSERT문에 호스트 변수, SQLDA 또는 널(NULL) 값만 사용해야 합니다.

대형 오브젝트(LOB) 데이터 유형

LOB 데이터 유형은 DB2 Connect에서 지원합니다.

사용자 정의 유형(UDT)

사용자 정의 특성 유형은 DB2 Connect에서만 지원합니다. 추상 데이터 유형은 지원되지 않습니다.

ROWID 데이터 유형

ROWID 데이터 유형은 DB2 Connect에서 2진 데이터용 VARCHAR로 처리합니다.

64비트 정수(BIGINT) 데이터 유형

8 바이트(64 비트) 정수는 DB2 Connect에서 지원됩니다. BIGINT 내부 데이터 유형은 데이터 정밀도를 유지하면서 아주 큰 데이터베이스의 기본 행수(cardinality)를 지원하는 데 사용합니다.

DCL(Data Control Language) 사용

각 IBM 관계형 데이터베이스 관리 시스템에서는 GRANT 및 REVOKE SQL문에 대해 다른 레벨의 granularity를 제공합니다. 특정 제품에 대한 책을 점검하여 각 데이터베이스 관리 시스템에 사용할 해당 SQL문을 검증하십시오.

연결 및 연결해제

DB2 Connect에서는 매개변수가 없는 CONNECT 외에 CONNECT문의 CONNECT TO 및 CONNECT RESET 버전을 지원합니다. 응용프로그램에서 먼저 명시적 CONNECT TO문을 수행하지 않고 SQL문을 호출할 경우 기본 응용프로그램 서버에 대한 *내재된* 연결이 수행됩니다(연결이 정의된 경우).

데이터베이스에 연결하면 관계형 데이터베이스 시스템을 식별하는 정보가 SQLCA의 SQLERRP 필드로 리턴됩니다. 응용프로그램 서버(AS)가 IBM 관계형 데이터베이스일 경우, SQLERRP의 처음 세 바이트에는 다음 중 하나가 포함됩니다.

DSN OS/390용 DB2 Universal Database

ARI VSE & VM용 DB2

QSQ AS/400용 DB2 Universal Database

SQL DB2 Universal Database.

DB2 Connect를 사용하는 동안 CONNECT TO 또는 널(NULL) CONNECT문을 발행할 경우 SQLCA의 SQLERRMC 필드에서 국가 코드 또는 지역 토큰은 공백으로 리턴됩니다. 응용프로그램 서버(AS)의 CCSID은 국가 코드 또는 코드 세트 토큰으로 리턴됩니다.

유형 1 연결의 경우 CONNECT RESET문을, 유형 2 연결의 경우 RELEASE 및 COMMIT문을, 연결 유형이지만 IP 모니터 환경이 아닐 경우 DISCONNECT문을 사용하여 명시적으로 연결해제할 수 있습니다.

연결을 명시적으로 연결해제하지 못하고 응용프로그램이 정상적으로 종료된 경우 DB2 Connect에서 결과 데이터를 내재적으로 확약합니다.

주: 응용프로그램에서는 오류와 여전히 정상적인 종료를 지시하는 SQLCODE를 수신할 수 있습니다. DB2 Connect에서는 이런 경우 데이터를 확약합니다. 데이터 확약을 원하지 않을 경우 ROLLBACK 명령을 발행해야 합니다.

FORCE 명령을 사용하여 데이터베이스에서 선택한 사용자나 모든 사용자를 연결해제할 수 있습니다. 호스트 또는 AS/400 서버 데이터베이스에서 이를 지원합니다. 사용자는 DB2 Connect 워크스테이션에서 강제로 분리될 수 있습니다.

사전 처리 컴파일

상이한 IBM 관계형 데이터베이스 시스템에 따라 사전 처리 컴파일러에 몇 가지 차이점이 있습니다. DB2 Universal Database용 사전 처리 컴파일러는 호스트 또는 AS/400 서버 사전 처리 컴파일러와 다음과 같은 점에서 다릅니다.

- 이 컴파일러는 응용프로그램을 한번만 통과합니다.
- DB2 Universal Database 데이터베이스에 대해 바인딩할 때 성공적인 바인드를 위해서는 오브젝트가 있어야 합니다. VALIDATE RUN은 지원되지 않습니다.

블로킹

DB2 Connect 프로그램에서는 DB2 데이터베이스 관리 프로그램 블로킹 바인드 옵션을 지원합니다.

UNAMBIG
불명확한 커서만 블로킹됩니다(기본값).

ALL 불명확한 커서가 블로킹됩니다.

NO 커서가 블로킹되지 않습니다.

DB2 Connect 프로그램에서는 RQRIOBLK 필드에 DB2 데이터베이스 관리 프로그램 구성 파일에서 정의한 블록 크기를 사용합니다. DB2 Connect의 현재 버전에서는 32 767까지 블록 크기를 지원합니다. DB2 데이터베이스 관리 프로그램에 더 큰 값이 지정된 경우 DB2 Connect에서는 32 767 중 한 값을 사용하지만 DB2 데이터베이스 관리 프로그램 구성 파일을 재설정하지는 않습니다. 블로킹은 동적 및 정적 SQL에 동일한 블록 크기를 사용할 경우와 같이 처리됩니다.

주: 대부분의 호스트 또는 AS/400 서버 시스템에서는 동적 커서를 불명확한 커서로 간주하지만 DB2 Universal Database 시스템에서는 몇 가지 동적 커서를 명확한 커서로 간주합니다. 혼동을 피하려면 DB2 Connect로 BLOCKING ALL을 지정할 수 있습니다.

Administrative API Reference 및 *Command Reference*에 나열된 CLP, 제어 센터 또는 API를 사용하여 DB2 데이터베이스 관리 프로그램 구성 파일에서 블록 크기를 지정하십시오.

패키지 속성

패키지에는 다음 속성이 있습니다.

콜렉션 ID
패키지의 ID. PREP 명령에서 지정할 수 있습니다.

소유자 패키지 소유자의 권한 부여 ID. PREP 또는 BIND 명령에서 지정할 수 있습니다.

작성자 패키지를 바인드하는 사용자 이름.

규정자 패키지의 오브젝트에 대한 내재된 규정자. PREP 또는 BIND 명령에서 지정할 수 있습니다.

각 호스트 또는 AS/400 서버 시스템의 경우 이런 속성 사용에 제한사항이 있습니다.

OS/390용 DB2 Universal Database
: 모두 네 개의 속성은 서로 다를 수 있습니다. 상이한 규정자를 사용하려면 특수한 관리 특권이 있어야 합니다. 이런 속성 사용과 관련된 조건에 대해서는 OS/390용 DB2 Universal Database용 *명령 참조서*를 참조하십시오.

VSE & VM용 DB2
: 모든 속성은 동일해야 합니다. USER1에서 PREP로 바인드 파일을 작성하고 USER2에서 실제 바인드를 수행할 경우 USER2에 USER1에 대해 바인드하는 DBA 권한이 있어야 합니다. 속성에는 USER1의 사용자 이름만 사용됩니다.

AS/400용 DB2 Universal Database
: 규정자는 콜렉션 이름을 지시합니다. 규정자와 소유권 사이의 관계는 오브젝트에 대한 특권의 권한 부여와 권한 취소에 영향을 줍니다. 콜렉션 ID에서 규정하지 않는 한 기록되는 사용자 이름은 작성자와 소유자입니다. 이 경우, 콜렉션 ID는 소유자입니다. 규정자에서 사용하기 전에 콜렉션 ID가 이미 있어야 합니다.

DB2 Universal Database
: 모두 네 개의 속성은 서로 다를 수 있습니다. 상이한 소유자를 사용하려면 관리자 권한이 필요하며 바인더에 스키마에 대한 CREATEIN 특권이 있어야 합니다(이미 존재할 경우).

주: DB2 Connect에서 OS/390용 DB2 Universal Database 및 DB2 Universal Database용 *SET CURRENT PACKAGESET* 명령을 지원합니다.

C 널로 종료되는 문자열

CNULREQD 바인드 옵션은 LANGLEVEL 옵션으로 지정한 널로 종료되는 문자열 처리를 대체합니다.

MIA 또는 SAA1로 설정된 LANGLEVEL 옵션으로 준비할 경우 널로 종료되는 문자열 처리 방법에 대해서는 689 페이지의 『C 및 C++의 널로 종료된 문자열』의 내용을 참조하십시오.

기본적으로 CNULREQD는 YES로 설정됩니다. 따라서 널로 종료되는 문자열은 MIA 표준에 따라 해석됩니다. OS/390용 DB2 Universal Database 서버에 연결할 경우 CNULREQD를 YES로 설정하는 것이 좋습니다. CNULREQD 옵션이 NO로 설정된 SAA1 표준으로 코딩된 응용프로그램에 바인드해야 합니다(널로 종료되는 문자열과 관련하여). 그렇지 않은 경우 SAA1로 설정된 LANGLEVEL을 사용하여 준비한 경우에도 MIA 표준에 따라 널로 종료되는 문자열은 해석됩니다.

독립형 SQLCODE 및 SQLSTATE

ISO/ANS SQL92에 정의된 독립형 SQLCODE 및 SQLSTATE 변수는 LANGLEVEL SQL92E 사전 처리 컴파일 옵션을 통해 지원됩니다. SQL0020W 경고는 사전 처리 컴파일 시간에 발행되어 LANGLEVEL이 지원되지 않음을 지시합니다. 이 경고는 LANGLEVEL SQL92E의 부속 집합인, *Command Reference*의 LANGLEVEL MIA 아래 나열된 기능에만 적용됩니다.

정렬 순서 정의

EBCDIC 및 ASCII 사이의 차이점은 다양한 데이터베이스 제품에서 다양한 정렬 순서를 낳으며 ORDER BY 및 GROUP BY절에도 영향을 줍니다. 이런 차이점을 최소화하는 한 가지 방법은 EBCDIC 정렬 순서를 모방하는 사용자 정의 조합 순서를 작성하는 것입니다. 새로운 데이터베이스를 작성할 때에만 조합 순서를 지정할 수 있습니다. 더 자세한 내용은 *Administrative API Reference* 및 *Command Reference*을 참조하십시오.

주: 이제 데이터베이스 테이블을 ASCII 형식으로 OS/390용 DB2 Universal Database에 저장할 수 있습니다. 따라서 DB2 Connect와 OS/390용 DB2 Universal Database 사이에서 데이터를 더 빠르게 교환할 수 있고 필드 프로시듀어를 제공할 필요가 없습니다. 그렇지 않은 경우에는 데이터를 변환하고 순서를 재지정하는 데 필드 프로시듀어를 사용해야 합니다.

참조 무결성 관리

시스템에 따라 참조 제한조건은 다르게 조절됩니다.

OS/390용 DB2 Universal Database
기본 키를 사용하여 외부 키를 작성하려면 먼저 기본 키에서 색인을 작성해야 합니다. 테이블은 자신을 참조할 수 있습니다.

VSE & VM용 DB2
외부 키에 대한 색인은 자동으로 작성됩니다. 테이블은 자신을 참조할 수 없습니다.

AS/400용 DB2 Universal Database
외부 키에 대한 색인은 자동으로 작성됩니다. 테이블은 자신을 참조할 수 있습니다.

DB2 Universal Database
DB2 Universal Database 데이터베이스의 경우 기본 키를 포함하여, 고유 제한조건에 대한 색인은 자동으로 작성됩니다. 테이블은 자신을 참조할 수 있습니다.

기타 규칙은 연쇄의 관련 레벨에 따라 달라집니다.

잠금

데이터베이스 서버에서 잠금을 수행하는 방법이 일부 응용프로그램에 영향을 줄 수 있습니다. 예를 들어, 행 레벨 잠금 및 커서 안정성(CS)의 분리 레벨 등으로 설계한 응용프로그램은 페이지 레벨 잠금을 수행하는 시스템으로 직접 이식할 수 없습니다. 이런 기본 차이점 때문에 응용프로그램을 조정해야 할 수 있습니다.

OS/390용 DB2 Universal Database 및 DB2 Universal Database 제품에는 잠금을 시간 종료하고 대기 중인 응용프로그램으로 오류 리턴 코드를 전송하는 기능이 있습니다.

SQLCODE와 SQLSTATE 사이의 차이점

상이한 IBM 관계형 데이터베이스 제품은 비슷한 오류에 대해 항상 동일한 SQLCODE를 생성하지 않습니다. 다음 두 방법 중 하나로 이런 문제를 처리할 수 있습니다.

- 특정 오류에 SQLCODE 대신 SQLSTATE를 사용하십시오.

  SQLSTATE는 여러 데이터베이스 제품 사이에서 대략 동일한 의미를 가지며 이 제품은 SQLCODE에 해당하는 SQLSTATE를 생성합니다.

- 한 시스템에서 다른 시스템으로 SQLCODE를 맵핑하십시오.

  기본적으로 DB2 Connect에서는 각 IBM 호스트 또는 AS/400 서버 시스템에서 DB2 Universal Database 시스템으로 SQLCODE 및 토큰을 맵핑합니다. 기본 맵핑을 대체하거나 SQLCODE 맵핑을 포함하지 않은 데이터베이스 서버(비-IBM 데이터베이스 서버)를 사용하지 않을 경우 사용자의 SQLCODE 맵핑 파일을 지정할 수 있습니다. 또한 SQLCODE 맵핑을 해제할 수 있습니다.

  자세한 정보는 *DB2 Connect 사용자 안내서* 책을 참조하십시오.

시스템 카탈로그 사용

시스템 카탈로그는 IBM 데이터베이스 제품에 따라 달라집니다. 뷰를 사용하여 많은 차이점을 마스킹할 수 있습니다. 자세한 내용은 사용 중인 데이터베이스 서버에 대한 설명서를 참조하십시오.

CLI의 카탈로그 함수는 DB2 계열에서 카탈로그 조회에 대해 동일 API 및 결과 세트 지원을 표시하여 이 문제를 해결합니다.

검색 할당에서 숫자 변환 오버플로우

검색 할당에서의 숫자 변환 오버플로우는 다른 IBM 관계형 데이터베이스 제품에 따라 다르게 처리될 수 있습니다. 예를 들어, OS/390용 DB2 Universal Database 및 DB2 Universal Database에서 정수 호스트 변수로 유동 컬럼을 가져올 수 있습니다. 유동 값을 정수 값으로 변환할 때 변환 오버플로우가 발생할 수 있습니다. 기본적으로 OS/390용 DB2 Universal Database에서 경고 SQLCODE 및 널(NULL) 값을 응용프로그램으로 리턴합니다. 반대로 DB2 Universal Database에

서는 변환 오버플로우 오류를 리턴합니다. 대략 크기를 지정한 호스트 변수로 가져와 응용프로그램은 검색 할당에서의 숫자 변환 오버플로우를 피하는 것이 좋습니다.

분리 레벨

응용프로그램을 준비하거나 바인드할 때 DB2 Connect에서는 다음과 같은 분리 레벨을 수용합니다.

RR 반복 읽기(RR)

RS 읽기 안정성(RS)

CS 커서 안정성(CS)

UR 미확약 읽기(UR)

NC 확약 안함

분리 레벨은 보호 레벨이 가장 높은 쪽에서 가장 낮은 순으로 나열됩니다. 호스트 또는 AS/400 서버에서 사용자가 지정한 분리 레벨을 지원하지 않을 경우 다음으로 높은 지원 레벨이 사용됩니다.

표55에서는 각 호스트 또는 AS/400 응용프로그램 서버(AS)에서의 각 분리 레벨의 결과를 나타냅니다.

표 55. 분리 레벨

DB2 Connect	OS/390용 DB2 Universal Database	VSE & VM용 DB2	AS/400용 DB2 Universal Database	DB2 Universal Database
RR	RR	RR	note 1	RR
RS	note 2	RR	COMMIT(*ALL)	RS
CS	CS	CS	COMMIT(*CS)	CS
UR	note 3	CS	COMMIT(*CHG)	UR
NC	note 4	note 5	COMMIT(*NONE)	UR

표 55. 분리 레벨 (계속)

DB2 Connect	OS/390용 DB2 Universal Database	VSE & VM용 DB2	AS/400용 DB2 Universal Database	DB2 Universal Database

주:

1. AS/400용 DB2 Universal Database에는 RR와 일치하는 해당 COMMIT 옵션이 없습니다. AS/400용 DB2 Universal Database에서는 전체 테이블을 잠금하여 RR을 지원합니다.
2. 버전 3.1의 경우 RR을 결과로 낳고, APAR PN75407이 포함된 버전 4.1 또는 버전 5.1의 경우 RS를 낳습니다.
3. 버전 3.1의 경우 CS를 결과로 낳고, 버전 4.1 또는 버전 5.1의 경우 UR을 낳습니다.
4. 버전 3.1의 경우 CS를 결과로 낳고, APAR PN60988이 포함된 버전 4.1 또는 버전 5.1의 경우 UR을 낳습니다.
5. VSE & VM용 DB2으로는 분리 레벨 NC를 지원하지 못합니다.

응용프로그램이 ALL로 설정된 블로킹 및 UR의 분리 레벨과 바인드되었거나 분리 레벨이 NC로 설정된 경우 AS/400용 DB2 Universal Database에서 기록되지 않는(unjournalled) 테이블에 액세스할 수 있습니다.

저장 프로시듀어

- 호출

  클라이언트 프로그램에서 SQL CALL문을 발행하여 서버 프로그램을 호출할 수 있습니다. 이 경우 각 서버는 다른 서버에 대해 약간 다르게 작업합니다.

  OS/390

  스키마 이름의 길이는 8 바이트여야 하며, 프로시듀어 이름의 길이는 18 바이트여야 하고, 저장 프로시듀어는 서버의 SYSIBM.SYSPROCEDURES 카탈로그에서 정의해야 합니다.

  VSE 또는 VM

  프로시듀어 이름의 길이가 18 바이트를 넘어선 안되며 그 이름은 서버의 SYSTEM.SYSROUTINES 카탈로그에서 정의해야 합니다.

  OS/400

  프로시듀어 이름은 SQL 식별자여야 합니다. 또한 DECLARE

PROCEDURE 또는 CREATE PROCEDURE문을 사용하여 실제 경로 이름(스키마 이름 또는 콜렉션 이름)을 지정해 저장 프로시듀어를 찾을 수 있습니다.

REXX/SQL에서 구현된 CALL문이 CALL USING DESCRIPTOR에 맵핑되므로 응용프로그램이 REXX/SQL에서 AS/400용 DB2로의 모든 CALL문은 동적으로 준비하여 실행해야 합니다.

SQL CALL문의 구문에 대해서는 *SQL 참조서*에서 자세한 내용을 참조하십시오.

서버 프로그램이 OS/390용 DB2 Universal Database, AS/400용 DB2 Universal Database 또는 VSE & VM용 DB2에서 사용한 것과 동일한 매개변수 규정으로 DB2 Universal Database에서 서버 프로그램을 호출할 수 있습니다. DB2 Universal Database 저장 프로시듀어 호출에 대한 자세한 내용은 207 페이지의 『제7장 저장 프로시듀어』를참조하십시오. 다른 플랫폼에서의 매개변수 규정에 대해서는 해당 플랫폼에 대한 DB2 제품 문서를 참조하십시오.

저장 프로시듀어의 모든 SQL문은 클라이언트 SQL 프로그램에서 시작한 SQL 작업 단위(UOW)로 실행됩니다.

- 저장 프로시듀어로 또는 저장 프로시듀어에서 특수한 의미를 가진 표시기 값을 전달하지 마십시오.

  DB2 Universal Database 사이에서 시스템은 사용자가 표시기 변수에 넣은 값을 전달합니다. 그러나 DB2 Connect를 사용할 경우 표시기 변수로 0, -1 및 -128만 전달할 수 있습니다.

- 서버 응용프로그램에서 발견한 오류나 경고를 리턴하도록 매개변수를 정의해야 합니다.

  DB2 Universal Database의 서버 프로그램에서 오류나 경고를 리턴하도록 SQLCA를 갱신할 수 있지만 OS/390용 DB2 Universal Database 또는 AS/400용 DB2 Universal Database의 저장 프로시듀어에서는 이를 지원하지 않습니다. 사용자의 저장 프로시듀어에서 오류 코드를 리턴하려면 매개변수로 전달해야 합니다. 서버는 시스템 탐지 오류를 위해 SQLCODE 및 SQLCA만 설정합니다.

- VSE & VM용 DB2 버전 7 이상 및 OS/390용 DB2 Universal Database 버전 5.1 이상은 이때 저장 프로시듀어의 결과 세트를 리턴할 수 있는 유일한 호스트 또는 AS/400 응용프로그램 서버(AS)입니다.

Stored Procedure Builder

DB2 Stored Procedure Builder는 저장 프로시듀어 작성, 설치 그리고 테스트에 대해 사용하기 쉬운 개발 환경을 제공합니다. 이를 통해 사용자는 DB2 서버에서 저장 프로시듀어 등록, 구축 그리고 설치에 대한 자세한 내용을 몰라도 저장 프로시듀어 논리 작성에 중점을 둘 수 있습니다. 또한, Stored Procedure Builder를 이용하여 한 운영 체제에서는 저장 프로시듀어를 개발하고, 다른 서버 운영 체제에서는 저장 프로시듀어를 구축할 수 있습니다.

Stored Procedure Builder는 신속한 개발을 지원하는 그래픽 응용프로그램입니다. Stored Procedure Builder를 사용하여 다음 타스크를 수행할 수 있습니다.

- 새로운 저장 프로시듀어 작성
- 지역 및 원격 DB2 서버에서 저장 프로시듀어 구축
- 기존의 저장 프로시듀어 수정 및 재구축
- 설치된 저장 프로시듀어 실행의 테스트 및 디버그

DB2 Universal Database 프로그램 그룹에서 Stored Procedure Builder를 별도의 응용프로그램으로 시작하거나 다음 개발 응용프로그램에서 Stored Procedure Builder를 시작할 수 있습니다.

- Microsoft Visual Studio
- Microsoft Visual Basic
- IBM VisualAge for Java

OS/390용 DB2의 제어 센터에서도 Stored Procedure Builder를 시작할 수 있습니다. Stored Procedure Builder를 제어 센터 도구 메뉴, 도구막대 또는 Stored Procedure 폴더에서 별도의 프로세스로 시작할 수 있습니다. 이 외에, Stored Procedure Builder 프로젝트 창에서 OS/390 서버용 DB2에 내장된 SQL 저장 프로시듀어를 하나 이상 명령행 프로세서(CLP)에서 실행될 수 있는 지정된 파일로 내보낼 수 있습니다.

Stored Procedure Builder는 프로젝트를 사용하여 작업을 관리합니다. 각 Stored Procedure Builder 프로젝트는 OS/390용 DB2 서버와 같은 특정 데이터베이스에 대한 연결을 저장합니다. 이 외에도, 필터를 작성하여 각 데이터베이스에서 저장 프로시듀어의 부속 집합을 표시할 수 있습니다. 신규 또는 기존 Stored Procedure Builder 프로젝트를 열 때 이름, 스키마, 언어 또는 모음 ID(OS/390만)를 기반으로 저장 프로시듀어를 볼 수 있도록 저장 프로시듀어를 필터할 수 있습니다.

연결 정보는 Stored Procedure Builder 프로젝트에 저장되므로 기존 프로젝트를 열면 데이터베이스에 대해 사용자 id와 암호를 입력하도록 자동 프롬프트됩니다. SQL 저장 프로시듀어 삽입 마법사를 사용하여 OS/390용 DB2 서버에서 SQL 저장 프로시듀어를 구축할 수 있습니다. OS/390 서버용 DB2에 구축된 SQL 저장 프로시듀어에 대해 특정 컴파일, 사전 링크, 바인드, 런타임, WLM 환경 그리고 외부 보안 옵션을 설정할 수 있습니다.

추가로 CPU 시간과 SQL 저장 프로시듀어가 실행 중인 스레드에 대한 기타 DB2 비용 정보를 포함하여 SQL 저장 프로시듀어에 대한 SQL 비용 정보를 얻을 수 있습니다. 특히, 래치/잠금 경합 대기 시간, getpage 수, 읽기 I/O 수 그리고 쓰기 I/O 수에 대한 비용 정보를 얻을 수 있습니다.

비용 정보를 얻으려면 Stored Procedure Builder는 OS/390 서버용 DB2에 연결하고 SQL문을 실행하며 저장 프로시듀어(DSNWSPM)를 호출하여 SQL 저장 프로시듀어에서 사용한 CPU 시간을 알아냅니다.

최소단위 복합 SQL이 아님

복합 SQL을 통해 복수 SQL문을 단일 실행 가능 블록으로 그룹화할 수 있습니다. 그렇게 하면 네트워크 오버헤드를 감소시키고 응답 시간을 향상시킬 수 있습니다.

DB2 Connect에서는 최소단위 복합 SQL이 아닌 SQL을 지원합니다. 즉, 복합 SQL 처리 뒤에 계속해서 오류가 발생합니다. DB2 Connect에서 지원하지 않는 최소단위 복합 SQL을 통해 오류는 복합 SQL의 전체 그룹을 구간 복원합니다.

응용프로그램 서버에서 종료할 때까지 명령문은 계속 실행됩니다. 일반적으로 복합 SQL문의 실행은 일련의 오류가 발생할 경우에만 중지됩니다.

최소단위 복합 SQL이 아닌 SQL은 지원되는 모든 호스트 또는 AS/400 응용프로그램 서버에 사용할 수 있습니다.

다중 SQL 오류가 발생할 경우, 처음 7개 실패 명령문은 SQLSTATE가 복수 오류가 발생했다는 메시지와 함께 SQLCA의 SQLERRMC 필드로 리턴됩니다. *SQL 참조서*에서 자세한 정보를 참조하십시오.

DB2 Connect와 다중 사이트 갱신

DB2 Connect를 통해 2단계 확약이라고도 하는 다중 사이트 갱신을 수행할 수 있습니다. 다중 사이트 갱신은 단일 분산 작업 단위(DUOW)에서 여러 데이터베이스를 갱신하는 것입니다. 몇 가지 인수에 따라 이 기능을 사용할 수 있는지 여부가 결정됩니다.

- 사용자의 응용프로그램은 CONNECT 2 및 SYNCPOINT TWOPHASE 옵션으로 사전 처리 컴파일해야 합니다.
- SNA 네트워크 연결을 사용하는 경우 AIX, OS/2 그리고 Windows NT에서 DB2 Connect Enterprise Edition 버전 7의 동기점 관리자 기능에서 제공하는 2단계 확약 지원을 사용할 수 있습니다. 이를 통해 다음 호스트 데이터베이스 서버는 분산 작업 단위에 참여할 수 있습니다.
  - AS/400용 DB2 버전 3.1 이상
  - MVS/ESA용 DB2 버전 3.1 이상
  - OS/390용 DB2 버전 5.1 이상
  - VM & VSE용 DB2 버전 5.1 이상

  위의 내용은 IBM TXSeries, CICS for Open Systems, BEA Tuxedo, Encina Monitor 그리고 Microsoft Transaction Server와 같은 외부 트랜잭션 처리(TP) 모니터로 조정되는 원시 DB2 UDB 응용프로그램과 응용프로그램에 대해 참입니다.

  주: BEA Tuxedo에 대한 자세한 정보는 *DB2 Connect 사용자 안내서*를 참조하십시오.
- TCP/IP 네트워크 연결이 있으면 OS/390용 DB2 V5.1 이상의 서버는 분산 작업 단위에 참여할 수 있습니다. 응용프로그램이 IBM TXSeries, CICS for Open

Systems, Encina Monitor 또는 Microsoft Transaction Server와 같은 트랜잭션 처리 모니터로 제어되면 도기지점 관리자를 사용해야 합니다.

원시 DB2 응용프로그램과 TP 모니터 응용프로그램에서 일반 DB2 Connect Enterprise Edition 서버를 사용하여 TCP/IP 연결을 통해 호스트 데이터를 액세스하면 동기점 관리자를 사용해야 합니다.

단일 DB2 Connect Enterprise Edition 서버를 사용하여 SNA 및 TCP/IP 네트워크 프로토콜을 사용하는 호스트 데이터를 액세스하고 두 단계 확약이 필요하면 동기점 관리자를 사용해야 합니다. 이는 DB2 응용프로그램과 TP 모니터 응용프로그램의 경우에도 마찬가지입니다.

DB2 Connect에서 지원하는 호스트 또는 AS/400 서버 SQL문

호스트 또는 AS/400 서버 처리의 경우 다음 명령문이 성공적으로 컴파일되지만 DB2 Universal Database 시스템의 처리의 경우에는 그렇지 않습니다.

- ACQUIRE
- DECLARE (modifier.(qualifier.)table_name TABLE ...
- LABEL ON

명령행 처리기에서도 이런 명령문을 지원합니다.

다음 명령문은 호스트 또는 AS/400 서버 처리에 대해 지원되지만 바인드 파일이나 패키지에 추가되지 않으며 명령행 처리기에서 지원하지도 않습니다.

- DESCRIBE statement_name INTO descriptor_name USING NAMES
- PREPARE statement_name INTO descriptor_name USING NAMES FROM ...

사전 처리 컴파일러에서는 다음과 같이 가정합니다.

- 호스트 변수는 입력 변수입니다.
- 명령문은 고유 절 번호에 할당됩니다.

DB2 Connect에서 거부하는 호스트 또는 AS/400 서버 SQL문

다음 SQL문은 DB2 Connect와 명령행 처리기에서 지원하지 않습니다.

- COMMIT WORK RELEASE
- DECLARE state_name, statement_name STATEMENT
- DESCRIBE statement_name INTO descriptor_name USING xxxx (where xxxx is ANY, BOTH, or LABELS)
- PREPARE statement_name INTO descriptor_name USING xxxx FROM :host_variable (where xxxx is ANY, BOTH, or LABELS)
- PUT ...
- ROLLBACK WORK RELEASE
- SET :host_variable = CURRENT ...

VSE & VM용 DB2 확장 동적 SQL문은 -104 및 구문 오류 SQLCODE로 거부됩니다.

부록E. EBCDIC 2진 조합 시뮬레이트

DB2를 사용하여, 사용자 정의 조합 순서에 따라 문자열을 조합할 수 있습니다. 이런 기능을 사용하여 EBCDIC 2진 조합을 시뮬레이트할 수 있습니다.

EBCDIC 조합을 시뮬레이트하는 한 예로, 코드 페이지 850으로 ASCII 데이터베이스를 작성하려 하지만 마치 데이터가 실제로 코드 페이지 500의 EBCDIC 데이터베이스 있는 것처럼 문자열을 조합하려 한다고 가정해봅시다. 코드 페이지 500의 정의에 대해서는 904 페이지의 그림26, 코드 페이지 850에 대해서는 905 페이지의 그림27에서 자세한 내용을 참조하십시오.

2진으로 조합할 때 EBCDIC 코드 페이지 500 데이터베이스에 있는 네 문자의 상대 조합을 고려해보십시오.

문자	**코드 페이지 500 코드 포인트**
'a'	X'81'
'b'	X'82'
'A'	X'C1'
'B'	X'C2'

코드 페이지 500 2진 조합 순서(원하는 순서)는 다음과 같습니다.

```
'a' < 'b' < 'A' < 'B'
```

ASCII 코드 페이지 850으로 데이터베이스를 작성할 경우 2진 조합은 다음과 같은 결과를 낳습니다.

문자	**코드 페이지 850 코드 포인트**
'a'	X'61'
'b'	X'62'
'A'	X'41'
'B'	X'42'

코드 페이지 850 2진 조합 순서(원하지 않는 순서)는 다음과 같습니다.

```
'A' < 'B' < 'a' < 'b'
```

원하는 순서를 얻으려면 사용자 정의 조합 순서로 데이터베이스를 작성해야 합니다. 이를 위해서는 샘플 조합 순서가 `sqle850a.h` include 파일로 DB2에 제공되어야 합니다. `sqle850a.h`의 내용은 그림25에 나와 있습니다.

```
#ifndef SQL_H_SQLE850A
#define SQL_H_SQLE850A

#ifdef __cplusplus
extern "C" {
#endif

unsigned char sqle_850_500[256] = {
0x00,0x01,0x02,0x03,0x37,0x2d,0x2e,0x2f,0x16,0x05,0x25,0x0b,0x0c,0x0d,0x0e,0x0f,
0x10,0x11,0x12,0x13,0x3c,0x3d,0x32,0x26,0x18,0x19,0x3f,0x27,0x1c,0x1d,0x1e,0x1f,
0x40,0x4f,0x7f,0x7b,0x5b,0x6c,0x50,0x7d,0x4d,0x5d,0x5c,0x4e,0x6b,0x60,0x4b,0x61,
0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0x7a,0x5e,0x4c,0x7e,0x6e,0x6f,
0x7c, 0xc1 , 0xc2 ,0xc3,0xc4,0xc5,0xc6,0xc7,0xc8,0xc9,0xd1,0xd2,0xd3,0xd4,0xd5,0xd6,
0xd7,0xd8,0xd9,0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0x4a,0xe0,0x5a,0x5f,0x6d,
0x79, 0x81 , 0x82 ,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x91,0x92,0x93,0x94,0x95,0x96,
0x97,0x98,0x99,0xa2,0xa3,0xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xc0,0xbb,0xd0,0xa1,0x07,
0x68,0xdc,0x51,0x42,0x43,0x44,0x47,0x48,0x52,0x53,0x54,0x57,0x56,0x58,0x63,0x67,
0x71,0x9c,0x9e,0xcb,0xcc,0xcd,0xdb,0xdd,0xdf,0xec,0xfc,0x70,0xb1,0x80,0xbf,0xff,
0x45,0x55,0xce,0xde,0x49,0x69,0x9a,0x9b,0xab,0xaf,0xba,0xb8,0xb7,0xaa,0x8a,0x8b,
0x2b,0x2c,0x09,0x21,0x28,0x65,0x62,0x64,0xb4,0x38,0x31,0x34,0x33,0xb0,0xb2,0x24,
0x22,0x17,0x29,0x06,0x20,0x2a,0x46,0x66,0x1a,0x35,0x08,0x39,0x36,0x30,0x3a,0x9f,
0x8c,0xac,0x72,0x73,0x74,0x0a,0x75,0x76,0x77,0x23,0x15,0x14,0x04,0x6a,0x78,0x3b,
0xee,0x59,0xeb,0xed,0xcf,0xef,0xa0,0x8e,0xae,0xfe,0xfb,0xfd,0x8d,0xad,0xbc,0xbe,
0xca,0x8f,0x1b,0xb9,0xb6,0xb5,0xe1,0x9d,0x90,0xbd,0xb3,0xda,0xfa,0xea,0x3e,0x41
};
#ifdef __cplusplus
}
#endif

#endif /* SQL_H_SQLE850A */
```

그림 25. 사용자 정의 조합 순서 - sqle_850_500

코드 페이지 850 문자에서 코드 페이지 500 2진 집합을 얻는 방법을 보려면 `sqle_850_500`의 샘플 조합 순서를 점검하십시오. 각 코드 페이지 850 문자에 대해 조합 순서에서의 그 가중치는 단지 코드 페이지 500의 해당 코드 포인트입니다.

예를 들어, 'a' 문자를 고려해보십시오. 905 페이지의 그림27과 같이 코드 페이지 850에서 이 문자의 코드 포인트는 X'61'입니다. `sqle_850_500` 배열에서 'a' 문자에는 X'81'의 가중치가 할당되었습니다(즉, `sqle_850_500` 배열의 98번째 구성요소).

위의 샘플 사용자 정의 조합 순서로 데이터베이스를 작성할 때 네 문자의 조합 방법을 고려해보십시오.

문자	코드 페이지 850 코드 포인트 / 가중치(sqle_850_500에서)
'a'	X'61' / X'81'
'b'	X'62' / X'82'
'A'	X'41' / X'C1'
'B'	X'42' / X'C2'

가중치에 따른 코드 페이지 850 사용자 정의 조합(원하는 조합)은 다음과 같습니다.

```
'a' < 'b' < 'A' < 'B'
```

이 예에서 원하는 수행을 시뮬레이트하도록 올바른 가중치를 지정하여 원하는 조합을 얻게 됩니다.

실제 조합 순서를 면밀히 관찰하면 순서 자체는 단순히 변환표라는 것을 알 수 있습니다. 이 변환표에서 소스 코드 페이지는 데이터 기반(850)의 코드 페이지이고 목표 코드 페이지는 원하는 2진 조합 코드 페이지(500)입니다. DB2에서 제공하는 다른 샘플 조합 순서를 사용하면 다른 변환을 얻을 수 있습니다. 사용자가 요구하는 변환표가 DB2에 있지 않을 경우 IBM 책 *Character Data Representation Architecture, Reference and Registry* SC09-2190에서 추가 변환표를 찾아볼 수 있습니다. 이 책과 함께 제공되는 CD-ROM에서 추가 변환표를 찾을 수 있습니다.

조합 순서에 대해서는 560 페이지의 『조합 순서 개요』에서 자세한 내용을 참조하십시오. 또한 DB2에 있는 조합 순서에 대해서, 그리고 사용자 정의 조합 순서로 데이터베이스를 작성하는 데모를 나타내는 샘플 프로그램의 목록(`db_udcs.c`)에 대해서는 *Administrative API Reference*에 설명된 CREATE DATABASE API를 참조하십시오.

16진값 첫번째➡ 두번째⬇	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0	(SP) SP010000	& SM030000	- SP100000	ø LO610000	Ø LO620000	° SM190000	µ SM170000	¢ SC040000	{ SM110000	} SM140000	\ SM070000	0 ND100000
-1	(RSP) SP300000	é LE110000	/ SP120000	É LE120000	a LA010000	j LJ010000	~ SD190000	£ SC020000	A LA020000	J LJ020000	÷ SA060000	1 ND010000
-2	â LA150000	ê LE150000	Â LA160000	Ê LE160000	b LB010000	k LK010000	s LS010000	¥ SC050000	B LB020000	K LK020000	S LS020000	2 ND020000
-3	ä LA170000	ë LE170000	Ä LA180000	Ë LE180000	c LC010000	l LL010000	t LT010000	· SD630000	C LC020000	L LL020000	T LT020000	3 ND030000
-4	à LA130000	è LE130000	À LA140000	È LE140000	d LD010000	m LM010000	u LU010000	© SM520000	D LD020000	M LM020000	U LU020000	4 ND040000
-5	á LA110000	í LI110000	Á LA120000	Í LI120000	e LE010000	n LN010000	v LV010000	§ SM240000	E LE020000	N LN020000	V LV020000	5 ND050000
-6	ã LA190000	î LI150000	Ã LA200000	Î LI160000	f LF010000	o LO010000	w LW010000	¶ SM250000	F LF020000	O LO020000	W LW020000	6 ND060000
-7	å LA270000	ï LI170000	Å LA280000	Ï LI180000	g LG010000	p LP010000	x LX010000	¼ NF040000	G LG020000	P LP020000	X LX020000	7 ND070000
-8	ç LC410000	ì LI130000	Ç LC420000	Ì LI140000	h LH010000	q LQ010000	y LY010000	½ NF010000	H LH020000	Q LQ020000	Y LY020000	8 ND080000
-9	ñ LN190000	ß LS610000	Ñ LN200000	` SD130000	i LI010000	r LR010000	z LZ010000	¾ NF050000	I LI020000	R LR020000	Z LZ020000	9 ND090000
-A	[SM060000	] SM080000	¦ SM650000	: SP130000	« SP170000	ª SM210000	¡ SP030000	¬ SM660000	(SHY) SP320000	¹ ND011000	² ND021000	³ ND031000
-B	. SP110000	$ SC030000	, SP080000	# SM010000	» SP180000	º SM200000	¿ SP160000	| SM130000	ô LO150000	û LU150000	Ô LO160000	Û LU160000
-C	< SA030000	* SM040000	% SM020000	@ SM050000	ð LD630000	æ LA510000	Đ LD620000	¯ SM150000	ö LO170000	ü LU170000	Ö LO180000	Ü LU180000
-D	(SP060000	) SP070000	_ SP090000	' SP050000	ý LY110000	¸ SD410000	Ý LY120000	¨ SD170000	ò LO130000	ù LU130000	Ò LO140000	Ù LU140000
-E	+ SA010000	; SP140000	> SA050000	= SA040000	þ LT630000	Æ LA520000	Þ LT640000	´ SD110000	ó LO110000	ú LU110000	Ó LO120000	Ú LU120000
-F	! SP020000	^ SD150000	? SP150000	" SP040000	± SA020000	¤ SC010000	® SM530000	× SA070000	õ LO190000	ÿ LY170000	Õ LO200000	(EO)

코드 페이지 501

그림 26. 코드 페이지 500

HEX DIGITS 1ST → 2ND ↓	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0		► SM590000	(SP) SP010000	0 ND100000	@ SM050000	P LP020000	` SD130000	p LP010000	Ç LC420000	É LE120000	á LA110000	░ SF140000	└ SF020000	ð LD630000	Ó LO120000	(SHY) SP320000
-1	☺ SS000000	◄ SM630000	! SP020000	1 ND010000	A LA020000	Q LQ020000	a LA010000	q LQ010000	ü LU170000	æ LA510000	í LI110000	▒ SF150000	┴ SF070000	Đ LD620000	ß LS610000	± SA020000
-2	☻ SS010000	↕ SM760000	" SP040000	2 ND020000	B LB020000	R LR020000	b LB010000	r LR010000	é LE110000	Æ LA520000	ó LO110000	▓ SF160000	┬ SF060000	Ê LE160000	Ô LO160000	‗ SM100000
-3	♥ SS020000	‼ SP330000	# SM010000	3 ND030000	C LC020000	S LS020000	c LC010000	s LS010000	â LA150000	ô LO150000	ú LU110000	│ SF110000	├ SF080000	Ë LE180000	Ò LO140000	¾ NF050000
-4	♦ SS030000	¶ SM250000	$ SC030000	4 ND040000	D LD020000	T LT020000	d LD010000	t LT010000	ä LA170000	ö LO170000	ñ LN190000	┤ SF090000	─ SF100000	È LE140000	õ LO190000	¶ SM250000
-5	♣ SS040000	§ SM240000	% SM020000	5 ND050000	E LE020000	U LU020000	e LE010000	u LU010000	à LA130000	ò LO130000	Ñ LN200000	Á LA120000	┼ SF050000	ı LI610000	Õ LO200000	§ SM240000
-6	♠ SS050000	▬ SM700000	& SM030000	6 ND060000	F LF020000	V LV020000	f LF010000	v LV010000	å LA270000	û LU150000	ª SM210000	Â LA160000	ã LA190000	Í LI120000	µ SM170000	÷ SA060000
-7	• SM570000	↨ SM770000	' SP050000	7 ND070000	G LG020000	W LW020000	g LG010000	w LW010000	ç LC410000	ù LU130000	º SM200000	À LA140000	Ã LA200000	Î LI160000	þ LT630000	¸ SD410000
-8	◘ SM570001	↑ SM320000	(SP060000	8 ND080000	H LH020000	X LX020000	h LH010000	x LX010000	ê LE150000	ÿ LY170000	¿ SP160000	© SM520000	╚ SF380000	Ï LI180000	Þ LT640000	° SM190000
-9	○ SM750000	↓ SM330000	) SP070000	9 ND090000	I LI020000	Y LY020000	i LI010000	y LY010000	ë LE170000	Ö LO180000	® SM530000	╣ SF230000	╔ SF390000	┘ SF040000	Ú LU120000	¨ SD170000
-A	◙ SM750002	→ SM310000	* SM040000	: SP130000	J LJ020000	Z LZ020000	j LJ010000	z LZ010000	è LE130000	Ü LU180000	¬ SM660000	║ SF240000	╩ SF400000	┌ SF010000	Û LU160000	· SD630000
-B	♂ SM280000	← SM300000	+ SA010000	; SP140000	K LK020000	[SM060000	k LK010000	{ SM110000	ï LI170000	ø LO610000	½ NF010000	╗ SF250000	╦ SF410000	█ SF610000	Ù LU140000	¹ ND011000
-C	♀ SM290000	∟ SA420000	, SP080000	< SA030000	L LL020000	\ SM070000	l LL010000	| SM130000	î LI150000	£ SC020000	¼ NF040000	╝ SF260000	╠ SF420000	▄ SF570000	ý LY110000	³ ND031000
-D	♪ SM930000	↔ SM780000	- SP100000	= SA040000	M LM020000	] SM080000	m LM010000	} SM140000	ì LI130000	Ø LO620000	¡ SP030000	¢ SC040000	═ SF430000	¦ SM650000	Ý LY120000	² ND021000
-E	♫ SM910000	▲ SM600000	. SP110000	> SA050000	N LN020000	^ SD150000	n LN010000	~ SD190000	Ä LA180000	× SA070000	« SP170000	¥ SC050000	╬ SF440000	Ì LI140000	¯ SM150000	■ SM470000
-F	☼ SM690000	▼ SV040000	/ SP120000	? SP150000	O LO020000	_ SP090000	o LO010000	⌂ SM790000	Å LA280000	ƒ SC070000	» SP180000	┐ SF030000	¤ SC010000	▀ SF600000	´ SD110000	(RSP) SP300000

Code Page 00850

그림 27. 코드 페이지 850

부록F. DB2 라이브러리 사용

DB2 Universal Database 라이브러리는 온라인 도움말, 책(PDF 및 HTML) 및 샘플 프로그램이 HTML 형식으로 구성됩니다. 이 절에서는 제공되는 정보 및 액세스하는 방법을 설명합니다.

제품 정보에 온라인으로 액세스하려면, 정보 센터를 사용하십시오. 922 페이지의 『정보 센터로 정보에 액세스』에서 자세한 내용을 참조하십시오. 사용자는 웹에서 타스크 정보, DB2 책, 문제점 해결 정보, 샘플 프로그램 및 DB2 정보를 볼 수 있습니다.

DB2 PDF 파일 및 인쇄된 책

DB2 정보

다음의 테이블은 DB2 책을 4개의 범주로 나눕니다.

DB2 안내 및 참조 정보

이 책에는 모든 플랫폼에 공통적인 DB2 정보가 들어 있습니다.

DB2 설치 및 구성 정보

이 책은 특정 플랫폼에서의 DB2에 대한 것입니다. 예를 들어, OS/2, Windows 및 UNIX 플랫폼에서의 DB2용으로 각각 다른 *빠른 시작* 책이 있습니다.

플랫폼간 샘플 프로그램(HTML)

이 샘플들은 응용프로그램 개발 클라이언트와 함께 설치된 샘플 프로그램의 HTML 버전입니다. 이들은 단지 정보용으로서 실제 프로그램을 대체하지는 않습니다.

릴리스 정보

이러한 파일에는 DB2 책에 포함될 수 없었던 최신 정보가 포함되어 있습니다.

© Copyright IBM Corp. 1993, 2000

설치 매뉴얼, 릴리스 정보 및 자습서는 제품 CD-ROM의 HTML 디렉토리에서 볼 수 있습니다. 대부분의 책은 단지 보기용으로 제품 CD-ROM에서 HTML 형식으로 제공되고 보기와 인쇄용으로 제품 CD-ROM에서 PDF 형식으로 제공됩니다. 또한 IBM에서 인쇄된 책을 주문하려면 917 페이지의『인쇄된 책 주문』에서 자세한 내용을 참조하십시오. 다음 테이블에는 주문할 수 있는 책을 보여줍니다.

OS/2 및 Windows 플랫폼에서는 sqllib\doc\html 디렉토리에 HTML 파일을 설치할 수 있습니다. DB2 정보는 여러 나라 언어로 번역되었습니다. 하지만, 모든 정보가 모든 나라의 언어로 번역된 것은 아닙니다. 정보가 특정 나라의 언어로 사용할 수 없을 경우에는 영문으로 제공됩니다.

UNIX 플랫폼에서는 doc/*%L*/html 디렉토리에 여러 나라 언어 버전의 HTML 파일을 설치할 수 있습니다. 여기서 *%L*은 해당 언어의 로케일을 나타냅니다. *빠른 시작* 책에서 보다 자세한 내용을 참조하십시오.

다음의 여러 가지 방법으로 DB2 책을 구하고 정보를 액세스할 수 있습니다.

- 921 페이지의『정보 온라인 보기』
- 925 페이지의『정보 온라인 검색』
- 917 페이지의『인쇄된 책 주문』
- 917 페이지의『PDF 책 인쇄』

표 56. DB2 정보

이름	설명	문서 번호 PDF 파일 이름	HTML 디렉토리
	DB2 안내 및 참조 정보		
관리 안내서	*관리 안내서: 계획*에서는 데이터베이스의 개념에 대한 개요, 논리적 또는 물리적인 데이터베이스 설계와 같은 설계에 대한 정보 그리고 고가용성에 대한 정보를 제공합니다. *관리 안내서: 구현*에서는 사용자의 설계, 데이터베이스 액세스, 감사, 백업 및 복구와 같은 구현에 대한 정보를 제공합니다. *관리 안내서: 성능*에서는 데이터베이스의 환경, 응용프로그램 성능 평가 및 성능 조정에 대한 정보를 제공합니다. 사용자는 문서 번호 SBOF-8934를 사용하여 세 권으로 된 *관리 안내서* 책을 주문할 수 있습니다.	SA30-0990 db2d1x70 SA30-0988 db2d2x70 SA30-0989 db2d3x70	db2d0
Administrative API Reference	데이터베이스를 관리하는 데 사용할 수 있는 DB2 API와 데이터 구조에 대해 설명합니다. 또한 응용프로그램에서 API를 호출하는 방법을 설명합니다.	SC09-2947 db2b0x70	db2b0
응용프로그램 빌드 안내서	환경 설정 정보를 제공하며, Windows, OS/2 및 UNIX 기반 플랫폼에서 DB2 응용프로그램을 컴파일, 링크 및 수행하는 단계별 지침을 제공합니다.	SA30-0991 db2axx70	db2ax
APPC, CPI-C, and SNA Sense Codes	DB2 Universal Database 제품을 사용할 때 접할 수 있는 APPC, CPI-C 및 SNA 감지 코드에 대한 일반 정보를 제공합니다. HTML 형식으로만 사용할 수 있습니다.	문서 번호가 없습니다. db2apx70	db2ap
응용프로그램 개발 안내서	Embedded SQL 또는 Java(JDBC 및 SQLJ)를 사용하여 DB2 데이터베이스를 액세스하는 응용프로그램을 개발하는 방법을 설명합니다. 저장 프로시듀어 작성, 사용자 정의 함수 작성, 사용자 정의 유형 작성, 트리거 사용, 파티션된 환경 또는 연합 시스템에서 응용프로그램을 개발하는 등의 다양한 주제가 다루어집니다.	SA30-0992 db2a0x70	db2a0

표 56. DB2 정보 (계속)

이름	설명	문서 번호 PDF 파일 이름	HTML 디렉토리
CLI Guide and Reference	DB2 Call Level Interface와 Microsoft ODBC 스펙과 호환 가능한 호출 가능 SQL 인터페이스를 사용하여 DB2 데이터베이스에 액세스하는 응용프로그램의 개발 방법에 대해 설명합니다.	SC09-2950 db2l0x70	db2l0
Command Reference	명령행 프로세서를 사용하는 방법을 설명하고 데이터베이스를 관리하기 위해 사용할 수 있는 DB2 명령을 설명합니다.	SC09-2951 db2n0x70	db2n0
연결성 보충 설명서	AS/400용 DB2, OS/390용 DB2, MVS용 DB2 또는 VM용 DB2를 DB2 Universal Database 서버와의 DRDA 응용프로그램 리퀘스터로 사용하는 방법에 대한 참조 정보 및 설치 정보를 제공합니다. 또한 DB2 Connect AR(응용프로그램 리퀘스터)과 함께 DRDA AS(응용프로그램 서버)를 사용하는 방법에 대해서도 상세히 설명합니다. HTML 및 PDF 형식으로만 사용할 수 있습니다.	문서 번호가 없습니다. db2h1x70	db2h1
데이터 이동 유틸리티 안내 및 참조서	Import, Export, Load, AutoLoader 및 DPROP와 같이 데이터 이동을 용이하게 해 주는 DB2 UDB 유틸리티의 사용 방법에 대해 설명합니다.	SA30-0994 db2dmx70	db2dm
Data Warehouse Center 관리 안내서	Data Warehouse Center를 사용하여 데이터 웨어하우스를 구축 및 유지보수하는 방법을 제공합니다.	SA30-1000 db2ddx70	db2dd
Data Warehouse Center 응용프로그램 통합 안내서	프로그래머들이 Data Warehouse Center 및 Information Catalog Manager를 응용프로그램과 통합하는 데 도움을 주는 정보를 제공합니다.	SA30-1001 db2adx70	db2ad
DB2 Connect 사용자 안내서	DB2 Connect 제품에 대한 개념, 프로그래밍 및 일반 사용 정보를 제공합니다.	SA30-0993 db2c0x70	db2c0
DB2 Query Patroller Administration Guide	DB2 Query Patroller 시스템의 조작 개요, 특정 조작 및 관리 정보, 관리 그래픽 사용자 인터페이스 유틸리티에 대한 타스크 정보를 제공합니다.	SC09-2958 db2dwx70	db2dw

표 56. DB2 정보 (계속)

이름	설명	문서 번호 **PDF 파일 이름**	**HTML 디렉토리**
DB2 Query Patroller User's Guide	DB2 Query Patroller의 도구 및 함수를 사용하는 방법을 설명합니다.	SC09-2960 db2wwx70	db2ww
용어집	DB2에서 사용되는 용어와 그 구성요소에 대한 정의를 제공합니다. HTML 형식과 *SQL 참조서*에서 사용할 수 있습니다.	문서 번호가 없습니다. db2t0x70	db2t0
Image, Audio 및 Video Extenders 관리 및 프로그래밍	DB2 Extender에 대한 일반적인 정보와 이미지, 오디오 및 비디오(IAV)의 관리 및 구성에 대한 정보 그리고 IAV extenders를 사용한 프로그램에 대한 정보를 제공합니다. 여기에는 참조 정보, 진단 정보(메시지 포함) 및 샘플도 들어 있습니다.	SA30-1043 dmbu7x70	dmbu7
Information Catalog Manager Administration Guide	정보 카탈로그 관리에 대한 지침을 제공합니다.	SC26-9995 db2dix70	db2di
Information Catalog Manager Programming Guide and Reference	Information Catalog Manager에 대한 아키텍처 인터페이스에 대한 정의를 제공합니다.	SC26-9997 db2bix70	db2bi
Information Catalog Manager 사용자 안내서	Information Catalog Manager 사용자 인터페이스 사용에 대한 정보를 제공합니다.	SA30-1002 db2aix70	db2ai
설치 및 구성 보충 설명서	플랫폼 특정 DB2 클라이언트의 플랜, 설치 및 설정에 대해 설명합니다. 또한 바인딩, 클라이언트 및 서버 통신의 설정, DB2 GUI 도구, DRDA AS, 분산 설치 및 이종 데이터 소스에 대한 분산 요구와 액세스 방식의 구성에 대한 정보가 들어 있습니다.	GA30-0975 db2iyx70	db2iy
메시지 참조서	DB2, Information Catalog Manager 및 Data Warehouse Center에서 발행하는 메시지와 코드를 나열하고 수행해야 할 조치에 대해 설명합니다. 문서 번호(SBOF-8932)를 사용하여 두 권으로 된 메시지 참조서 책을 모두 주문할 수 있습니다.	볼륨 1 GA30-0986 db2m1x70 볼륨 2 GA30-0987 db2m2x70	db2m0

표 56. DB2 정보 (계속)

이름	설명	문서 번호 PDF 파일 이름	HTML 디렉토리
OLAP Integration Server Administration Guide	OLAP 통합 서버의 관리 프로그램 구성요소를 사용하는 방법을 설명합니다.	SC27-0787 db2dpx70	n/a
OLAP Integration Server Metaoutline User's Guide	표준 OLAP Metaoutline 인터페이스 (Metaoutline Assistant가 아닌)를 사용하여 OLAP Metaoutlines을 작성하고 사용하는 방법을 설명합니다.	SC27-0784 db2upx70	n/a
OLAP Integration Server Model User's Guide	표준 OLAP 모델 인터페이스(Model Assistant 가 아닌)를 사용하여 OLAP 모델을 작성하는 방법을 설명합니다.	SC27-0783 db2lpx70	n/a
OLAP 설치 및 사용자 안내서	OLAP Starter Kit에 대한 구성 및 설치 정보를 제공합니다.	SA30-1074 db2ipx70	db2ip
Excel용 OLAP Spreadsheet Add-in 사용자 안내서	Excel 스프레드시트 프로그램을 사용하여 OLAP 데이터를 분석하는 방법을 설명합니다.	SA30-0564 db2epx70	db2ep
Lotus 1-2-3용 OLAP Spreadsheet Add-in 사용자 안내서	Lotus 1-2-3 스프레드시트 프로그램을 사용하여 OLAP 데이터를 분석하는 방법을 설명합니다.	SA30-0565 db2tpx70	db2tp
복제 안내 및 참조서	DB2와 함께 제공되는 IBM 복제 도구에 대한 플랜, 구성, 관리 및 사용 정보를 제공합니다.	SA30-1003 db2e0x70	db2e0
Spatial Extender 사용자 안내 및 참조서	Spatial Extender 설치, 구성, 관리, 프로그래밍 및 문제 해결에 대한 정보를 제공합니다. 또한 공간 데이터 개념에 대한 설명을 제공하고 Spatial Extender에만 고유하게 적용되는 참조 정보(메시지 및 SQL)를 제공합니다.	SA30-1045 db2sbx70	db2sb
SQL 시작하기	SQL 개념을 소개하고, 많은 구조와 타스크에 관한 예를 보여줍니다.	SA30-0996 db2y0x70	db2y0
SQL 참조서, 볼륨 1 및 볼륨 2	SQL 구문, 의미 및 언어 규칙을 설명합니다. 또한 릴리스 간 비호환성, 제품 제한사항 및 카탈로그 뷰에 대한 정보도 들어 있습니다. SBOF-8933 문서 번호를 사용하여 *SQL 참조서*를 주문할 수 있습니다.	볼륨 1 SA30-0997 db2s1x70 볼륨 2 SA30-0998 db2s2x70	db2s0

표 56. DB2 정보 (계속)

이름	설명	문서 번호 PDF 파일 이름	HTML 디렉토리
시스템 모니터 안내 및 참조서	데이터베이스와 데이터베이스 관리 프로그램에 관한 다양한 종류의 정보를 수집하는 방법을 설명합니다. 이 책은 데이터베이스 활동을 이해하고, 성능을 향상시키고, 문제점의 원인을 판별하기 위한 정보를 사용하는 방법을 설명합니다.	SA30-0995 db2f0x70	db2f0
Text Extender 관리 및 프로그래밍	DB2 extenders에 관한 일반적인 정보와 Text extenders 관리 및 구성에 관한 정보, Text extenders를 사용한 프로그래밍에 관한 정보를 제공합니다. 여기에는 참조 정보, 진단 정보(메시지 포함) 및 샘플도 들어 있습니다.	SA30-1044 desu9x70	desu9
문제점 해결 안내서	오류의 출처를 판별하고 문제점으로부터 회복하고, DB2 고객 서비스와 상담하여 진단 도구를 사용하는 것을 도와줍니다.	GA30-0704 db2p0x70	db2p0
새로운 기능	DB2 Universal Database, 버전 7의 새로운 특성, 기능 및 향상된 내용을 설명합니다.	SA30-0999 db2q0x70	db2q0
	DB2 설치 및 구성 정보		
OS/2 및 Windows용 DB2 Connect Enterprise Edition 빠른 시작	OS/2 및 Windows 32 비트 운영 체제에서 DB2 Connect Enterprise Edition에 관한 플랜, 설치, 이주 및 구성 정보를 제공합니다. 또한 지원되는 많은 클라이언트에 대한 설치 및 설정 정보도 들어 있습니다.	GA30-0974 db2c6x70	db2c6
UNIX용 DB2 Connect Enterprise Edition 빠른 시작	UNIX 기반 플랫폼에서의 DB2 Connect Enterprise Edition에 대한 플랜, 이주, 설치, 구성 및 타스크 정보를 제공합니다. 또한 지원되는 많은 클라이언트에 대한 설치 및 설정 정보도 들어 있습니다.	GA30-0973 db2cyx70	db2cy
DB2 Connect Personal Edition 빠른 시작	OS/2 및 Windows 32 비트 운영 체제에서 DB2 Connect Personal Edition에 관한 플랜, 설치, 이주 및 구성 정보를 제공합니다. 또한 지원되는 모든 클라이언트에 대한 설치 및 설정 정보도 들어 있습니다.	GA30-0981 db2c1x70	db2c1
DB2 Connect Personal Edition Quick Beginnings for Linux	지원되는 모든 Linux에서 DB2 Connect Personal Edition에 관한 플랜, 설치, 이주 및 구성 정보를 제공합니다.	GC09-2962 db2c4x70	db2c4

표 56. DB2 정보 (계속)

이름	설명	문서 번호 PDF 파일 이름	HTML 디렉토리
DB2 Data Links Manager 빠른 시작	AIX 및 Windows 32 비트 운영 체제용 DB2 Data Links Manager에 대한 플랜, 설치, 구성 및 타스크 정보를 제공합니다.	GA30-0980 db2z6x70	db2z6
UNIX용 DB2 Enterprise - Extended Edition 빠른 시작	UNIX 기반 플랫폼에서의 DB2 Enterprise - Extended Edition 플랜, 설치 및 구성 정보를 제공합니다. 또한 지원되는 많은 클라이언트에 대한 설치 및 설정 정보도 들어 있습니다.	GA30-0978 db2v3x70	db2v3
Windows용 DB2 Enterprise - Extended Edition 빠른 시작	Windows 32 비트 운영 체제용 DB2 Enterprise - Extended Edition에 관한 플랜, 설치 및 구성 정보를 제공합니다. 또한 지원되는 많은 클라이언트에 대한 설치 및 설정 정보도 들어 있습니다.	GA30-0977 db2v6x70	db2v6
OS/2용 DB2 빠른 시작	OS/2 운영 체제에서의 DB2 Universal Database에 관한 플랜, 설치, 이주 및 구성 정보를 제공합니다. 또한 지원되는 많은 클라이언트에 대한 설치 및 설정 정보도 들어 있습니다.	GA30-0982 db2i2x70	db2i2
UNIX용 DB2 빠른 시작	UNIX 기반 플랫폼에서의 DB2 Universal Database에 관한 플랜, 설치, 이주 및 구성 정보를 제공합니다. 또한 지원되는 많은 클라이언트에 대한 설치 및 설정 정보도 들어 있습니다.	GA30-0984 db2ixx70	db2ix
Windows용 DB2 빠른 시작	Windows 32 비트 운영 체제에서 DB2 Universal Database에 관한 플랜, 설치, 이주 및 구성 정보를 제공합니다. 또한 지원되는 많은 클라이언트에 대한 설치 및 설정 정보도 들어 있습니다.	GA30-0985 db2i6x70	db2i6
DB2 Personal Edition 빠른 시작	OS/2 및 Windows 32 비트 운영 체제에서의 DB2 Universal Database Personal Edition에 관한 플랜, 설치, 이주 및 구성 정보를 제공합니다.	GA30-0983 db2i1x70	db2i1
DB2 Personal Edition Quick Beginnings for Linux	지원되는 모든 Linux에서 DB2 Universal Database Personal Edition에 관한 플랜, 설치, 이주 및 구성 정보를 제공합니다.	GC09-2972 db2i4x70	db2i4
DB2 Query Patroller 설치 안내서	DB2 Query Patroller에 관한 설치 정보를 제공합니다.	GA30-0976 db2iwx70	db2iw

표 56. DB2 정보 (계속)

이름	설명	문서 번호 PDF 파일 이름	HTML 디렉토리
DB2 Warehouse Manager 설치 안내서	웨어하우스 에이전트, 웨어하우스 변환기 및 Information Catalog Manager에 관한 설치 정보를 제공합니다.	GA30-1027 db2idx70	db2id
	플랫폼간 샘플 프로그램(HTML)		
샘플 프로그램(HTML)	DB2가 지원하는 모든 플랫폼에서 프로그래밍 언어에 대한 샘플 프로그램이 HTML 형식으로 제공됩니다. 이 샘플 프로그램은 정보용으로만 제공됩니다. 모든 샘플을 모든 프로그래밍 언어로 사용할 수 있는 것은 아닙니다. HTML 샘플은 DB2 응용프로그램 개발 클라이언트가 설치될 때에 사용할 수 있습니다. 프로그램에 대한 정보는 *응용프로그램 빌드 안내서*에서 자세한 내용을 참조하십시오.	문서 번호가 없습니다.	db2hs
	릴리스 정보		
DB2 Connect 릴리스 정보	DB2 Connect 책에는 포함될 수 없었던 최신 정보를 제공합니다.	#2를 참조하십시오.	db2cr
DB2 설치 정보	DB2 책에는 포함될 수 없었던 최신 설치 정보를 제공합니다.	제품 CD-ROM에서만 사용할 수 있습니다.	
DB2 릴리스 정보	DB2 책에는 포함될 수 없었던 모든 DB2 제품 및 기능에 대한 최신 정보를 제공합니다.	#2를 참조하십시오.	db2ir

주:

1. 파일 이름의 6번째 자리에 있는 문자 *x*는 책의 언어 버전을 나타냅니다. 예를 들면, 파일 이름 `db2d0e70`은 *관리 안내서* 책의 영문 버전을 나타내며 `db2d0k70`은 같은 책의 한글 버전을 나타냅니다. 다음 문자는 언어 버전을 나타내기 위해 파일 이름의 6번째 자리에 사용됩니다.

언어	식별자
브라질 포르투갈어	b
불가리아어	u
체코어	x
덴마크어	d
네덜란드어	q
영어	e

핀란드어	y
프랑스어	f
독일어	g
그리스어	a
헝가리어	h
이탈리아어	i
일본어	j
한글	k
노르웨이어	n
폴란드어	p
포르투갈어	v
러시아어	r
중국어	c
슬로베니아어	l
스페인어	z
스웨덴어	s
대만어	t
터키어	m

2. DB2 책에 포함되어 있지 않을 수 있는 최신 정보는 릴리스 정보에서 HTML 형식과 ASCII 파일로 사용할 수 있습니다. HTML 버전은 정보 센터와 제품 CD-ROM에서 사용할 수 있습니다. ASCII 파일을 보려면,
   - UNIX 기반 플랫폼의 경우에는 Release.Notes 파일을 참조하십시오. 이 파일은 DB2DIR/Readme/*%L* 디렉토리에 있으며 여기서, *%L*은 로케일 이름이고 DB2DIR은 다음과 같습니다.
     - AIX에서는 /usr/lpp/db2_07_01
     - HP-UX, PTX, Solaris 및 Silicon Graphics IRIX에서는 /opt/IBMdb2/V7.1
     - Linux에서는 /usr/IBMdb2/V7.1
   - 다른 플랫폼의 경우에는 RELEASE.TXT 파일을 참조하십시오. 이 파일은 제품이 설치된 디렉토리에 있습니다. OS/2 플랫폼에서는 **IBM DB2** 폴더를 더블 클릭하고 **릴리스 정보** 아이콘을 더블 클릭할 수 있습니다.

PDF 책 인쇄

책의 사본을 원하는 경우 DB2 책 CD-ROM에 있는 PDF 파일을 인쇄할 수 있습니다. Adobe Acrobat Reader를 사용하여 책 전체나 특정 페이지를 인쇄할 수 있습니다. 라이브러리에 있는 각 책의 파일 이름에 대해서는 909 페이지의 표56에서 자세한 내용을 참조하십시오.

Adobe 웹 사이트인 http://www.adobe.com에서 Adobe Acrobat Reader의 최신 버전을 얻을 수 있습니다.

PDF 파일은 파일 확장자가 PDF로서 DB2 책 CD-ROM에 들어 있습니다. PDF 파일을 액세스하려면,

1. DB2 책 CD-ROM을 삽입하십시오. UNIX 기반의 플랫폼에서는 DB2 책 CD-ROM을 마운트해야 합니다. 마운트 절차에 대해서는 *빠른 시작* 책에서 자세한 내용을 참조하십시오.
2. Acrobat Reader를 시작하십시오.
3. 다음 위치에서 원하는 PDF 파일을 여십시오.
   - OS/2 및 Windows 플랫폼에서:

     x:\doc*language* 디렉토리. 여기서 *x*는 CD-ROM 드라이브를 나타내며 *language*는 사용자 언어를 나타내는 2문자 국가 코드를 나타냅니다. 예를 들면 영문인 경우에는 EN입니다.
   - UNIX 기반 플랫폼에서:

     /cdrom/doc/*%L* 디렉토리. 여기서 */cdrom*은 CD-ROM의 마운트 위치이고 *%L*은 원하는 로케일의 이름입니다.

또한 PDF 파일을 CD-ROM에서 지역이나 네트워크로 파일을 복사하고 거기서 읽을 수도 있습니다.

인쇄된 책 주문

인쇄된 DB2 책은 책 주문 번호(SBOF)를 사용하여 세트나 낱권으로 주문할 수 있습니다. 인쇄본을 주문하려면, IBM 협력업체 또는 영업 대표에게 문의하십시오. 또한 웹 사이트 http://www.elink.ibmlink.ibm.com/pbl/pbl에서도 책을 주문할 수 있습니다.

두 종류의 책 세트를 사용할 수 있습니다. SBOF-8935는 DB2 Warehouse Manager에 대한 참조 및 사용에 관한 정보를 제공합니다. SBOF-8931은 다른 모든 DB2 Universal Database 제품과 특징에 대한 참조 및 사용 정보를 제공합니다. 각 SBOF의 내용은 다음 테이블에 나열되어 있습니다.

표 57. 인쇄된 책 주문

SBOF 번호	포함된 책	
SBOF-8931	• 관리 안내서: 계획 • 관리 안내서: 구현 • 관라 안내서: 성능 • Administrative API Reference • 응용프로그램 빌드 안내서 • 응용프로그램 개발 안내서 • CLI Guide and Reference • Command Reference • 데이터 이동 유틸리티 안내 및 참조서 • Data Warehouse Center 관리 안내서 • Data Warehouse Center 응용프로그램 통합 안내서 • DB2 Connect 사용자 안내서 • 설치 및 구성 보충 설명서 • Image, Audio 및 Video Extenders 관리 및 프로그래밍 • 메시지 참조서, 볼륨 1 및 2	• OLAP Integration Server Administration Guide • OLAP Integration Server Metaoutline User's Guide • OLAP Integration Server Model User's Guide • OLAP Integration Server User's Guide • OLAP 설치 및 사용자 안내서 • Excel용 OLAP Spreadsheet Add-in 사용자 안내서 • Lotus 1-2-3용 OLAP Spreadsheet Add-in 사용자 안내서 • 복제 안내 및 참조서 • Spatial Extender 관리 및 프로그래밍 안내서 • SQL 시작하기 • SQL 참조서, 볼륨 1 및 2 • 시스템 모니터 안내 및 참조서 • Text Extender 관리 및 프로그래밍 • 문제점 해결 안내서 • 새로운 기능
SBOF-8935	• Information Catalog Manager Administration Guide • Information Catalog Manager 사용자 안내서 • Information Catalog Manager Programming Guide and Reference	• Query Patroller Administration Guide • Query Patroller User's Guide

DB2 온라인 문서

온라인 도움말 액세스

온라인 도움말은 모든 DB2 구성요소에서 사용할 수 있습니다. 다음의 테이블에서는 다양한 도움말 유형을 설명합니다.

도움말의 유형	내용	액세스하는 방법
명령 도움말	명령행 처리기 내의 명령 구문을 설명합니다.	대화식 모드인 명령행 처리기에서, 다음을 입력하십시오. `? command` 여기서, *command*는 키워드이거나 전체 명령입니다. 예를 들어, `? catalog`는 모든 CATALOG 명령에 대한 도움말을 표시하고, `? catalog database`는 CATALOG DATABASE 명령에 대한 도움말을 표시합니다.
클라이언트 구성 지원 프로그램 도움말 *명령 센터 도움말* *제어 센터 도움말* *Data Warehouse Center 도움말* *이벤트 분석기 도움말* *Information Catalog Manager 도움말* *위성 관리 센터 도움말* *스크립트 센터 도움말*	창 또는 노트북에서 수행할 수 있는 타스크를 설명합니다. 도움말은 알아야 할 개요와 전제조건 정보를 포함하고, 창 또는 노트북 제어를 사용하는 방법을 설명합니다.	창이나 노트북에서, **도움말** 버튼을 누르거나 **F1** 키를 누르십시오.

도움말의 유형	내용	액세스하는 방법
메시지 도움말	메시지의 원인과 사용자가 취해야 할 조치를 설명합니다.	대화식 모드인 명령행 처리기에서, 다음을 입력하십시오. `? XXXnnnnn` 여기서, *XXXnnnnn*은 유효한 메시지 식별자입니다. 예를 들어, `? SQL30081`은 SQL30081 메시지에 대한 도움말을 표시합니다. 한 번에 한 화면씩 메시지 도움말을 보려면, 다음을 입력하십시오. `? XXXnnnnn	more` 파일에 메시지 도움말을 저장하려면, 다음을 입력하십시오. `? XXXnnnnn > filename.ext` 여기서, *filename.ext*는 메시지 도움말을 저장하려는 파일입니다.
SQL 도움말	SQL문의 구문을 설명합니다.	대화식 모드인 명령행 처리기에서, 다음을 입력하십시오. `help statement` 여기서, *statement*는 SQL문입니다. 예를 들어, help SELECT는 SELECT문에 대한 도움말을 표시합니다. **주:** SQL 도움말은 UNIX 기반 플랫폼에서 사용할 수 없습니다.
SQLSTATE 도움말	SQL문과 클래스 코드를 설명하십시오.	대화식 모드인 명령행 처리기에서, 다음을 입력하십시오. `? sqlstate` 또는 `? class code` 여기서, *sqlstate*는 유효한 5자리 숫자로 된 SQL 상태이고 *class code*는 SQL 상태의 처음 2자리 숫자입니다. 예를 들어, `? 08003`은 08003 SQL 상태에 대한 도움말을 표시하고, `? 08`은 08 클래스 코드에 대한 도움말을 표시합니다.

정보 온라인 보기

이 제품에 들어 있는 책은 HTML(Hypertext Markup Language) 소프트카피 형식으로 제공됩니다. 소프트카피는 정보를 검색할 수 있게 하고 관련된 정보로 링크하는 하이퍼텍스트를 제공합니다. 또한, 사이트에서 라이브러리를 공유하는 것도 더 쉬워집니다.

HTML 버전 3.2 스펙을 따르는 브라우저로 온라인 책 또는 샘플 프로그램을 볼 수 있습니다.

온라인 책 또는 샘플 프로그램을 보려면:

- DB2 관리 도구를 수행할 경우, 정보 센터를 사용하십시오.
- 브라우저에서, **파일 —>페이지 열기**를 클릭하십시오. 열린 페이지에 DB2 정보에 대한 설명과 링크가 들어 있습니다.
  - UNIX 기반 플랫폼에서는 다음과 같은 페이지를 여십시오.

    INSTHOME/sqllib/doc/*%L*/html/index.htm

    여기서 *%L*은 로케일 이름입니다.
  - 다른 플랫폼에서는 다음과 같은 페이지를 여십시오.

    sqllib\doc\html\index.htm

    이 경로는 DB2가 설치된 드라이브에 있습니다.

    정보 센터를 설치하지 않은 경우, **DB2 정보** 아이콘을 더블 클릭하여 페이지를 열 수 있습니다. 사용하는 시스템에 따라, 주 제품 폴더나 Windows 시작 메뉴에 아이콘이 있습니다.

Netscape 브라우저 설치

웹 브라우저를 설치하지 않은 경우, 제품 상자에 있는 Netscape CD-ROM에서 Netscape를 설치할 수 있습니다. 설치하는 방법에 대한 자세한 지시 사항은 다음을 수행하십시오.

1. Netscape CD-ROM을 삽입하십시오.
2. UNIX 기반의 플랫폼에서는 CD-ROM을 마운트해야 합니다. 마운트 절차에 대해서는 *빠른 시작* 책에서 자세한 내용을 참조하십시오.

3. 설치 지침서는 CDNAV*nn*.txt 파일을 참조하십시오. 여기서, *nn*은 2문자로 된 언어 식별자입니다. 파일은 CD-ROM의 루트 디렉토리에 있습니다.

정보 센터로 정보에 액세스

정보 센터는 DB2 제품 정보로의 빠른 액세스를 제공합니다. 정보 센터는 DB2 관리 도구를 사용할 수 있는 모든 플랫폼에서 사용할 수 있습니다.

정보 센터 아이콘을 더블 클릭하여 정보 센터를 열 수 있습니다. 사용하는 시스템에 따라 아이콘은 주 제품 폴더나 Windows **시작** 메뉴의 정보 폴더에 있습니다.

또한 DB2 Windows 플랫폼에서 도구 모음이나 **도움말** 메뉴를 사용하여 정보 센터를 액세스할 수 있습니다.

정보 센터는 6개 유형의 정보를 제공합니다. 적당한 탭을 클릭하여 그 유형에서 지원하는 주제를 보십시오.

타스크
DB2를 사용하여 수행할 수 있는 키 타스크.

참조
키워드, 명령 및 API와 같은 DB2 참조 정보.

책
DB2 책.

문제점 해결
오류 메시지의 종류와 복구 조치.

샘플 프로그램
DB2 응용프로그램 개발 클라이언트와 함께 제공되는 샘플 프로그램. DB2 응용프로그램 개발 클라이언트를 설치하지 않은 경우, 이 탭은 표시되지 않습니다.

웹
월드 와이드 웹에서의 DB2 정보. 이 정보에 액세스하려면, 사용자의 시스템으로부터 웹으로의 연결이 있어야 합니다.

목록 중 하나에서 항목을 선택할 때, 정보 센터는 정보를 표시하기 위해 표시기를 시작합니다. 표시기는 사용자가 선택하는 정보의 종류에 따라, 시스템 도움말 표시기, 편집기 또는 웹브라우저가 될 수 있습니다.

정보 센터는 찾기 기능을 제공하므로 목록을 찾지 않고도 특정 주제를 찾을 수 있습니다.

전체 텍스트 검색을 위해서는 **DB2 온라인 정보 검색** 검색 양식으로 연결된 정보 센터의 하이퍼텍스트 링크를 따라 검색하십시오.

HTML 검색 서버는 보통 자동으로 시작됩니다. HTML 정보에서 검색 기능이 작동하지 않으면, 다음 방법 중 하나를 사용하여 검색 서버를 시작할 수 있습니다.

Windows의 경우:
: **시작**을 클릭하고 **프로그램 —> IBM DB2 —> 정보 —> HTML 검색 서버 시작**을 선택하십시오.

OS/2 경우:
: **OS/2용 DB2** 폴더를 더블 클릭하고 **HTML 검색 서버 시작** 아이콘을 더블 클릭하십시오.

HTML 정보를 검색하면서 다른 문제가 생길 경우, 릴리스 정보를 참조하십시오.

주: 검색 기능은 Linux, PTX 및 Silicon Graphics IRIX 환경에서는 작동하지 않습니다.

DB2 마법사 사용

마법사는 한 번에 한 단계씩 각 작업을 수행하게 함으로써 특정 관리 타스크를 완료하는 데 도움을 줍니다. 마법사는 제어 센터 및 클라이언트 구성 지원 프로그램을 통해 사용할 수 있습니다. 다음 테이블에서는 마법사를 나열하고 그 기능을 설명합니다.

주: 데이터베이스 작성, 색인 작성, 다중 사이트 갱신 구성 및 성능 구성 마법사는 파티션된 데이터베이스 환경에서 사용할 수 있습니다.

마법사	도움 대상	액세스 방법
데이터베이스 추가	클라이언트 워크스테이션의 데이터베이스를 카탈로그화합니다.	클라이언트 구성 지원 프로그램에서 **추가**를 클릭하십시오.
데이터베이스 백업	백업 계획을 결정하고, 작성하고, 일정을 세웁니다.	제어 센터에서 백업하려는 데이터베이스를 마우스의 오른쪽 버튼으로 클릭한 다음 **백업 —> 마법사를 사용한 데이터베이스 백업**을 선택하십시오.
다중 사이트 갱신 구성	다중 사이트 갱신, 분산 트랜잭션 또는 2 단계 확약을 구성합니다.	제어 센터에서 **데이터베이스** 폴더를 마우스의 오른쪽 버튼으로 클릭하고 **다중 사이트 갱신**을 선택하십시오.

마법사	도움 대상	액세스 방법
데이터베이스 작성	데이터베이스를 작성한 다음, 몇 가지 기본적인 구성 타스크를 수행합니다.	제어 센터에서 **데이터베이스** 폴더를 마우스의 오른쪽 버튼으로 클릭한 다음 **작성 —> 마법사를 사용한 데이터베이스 작성**을 선택하십시오.
테이블 작성	기본 데이터 유형을 선택한 다음, 테이블에 대한 기본 키를 작성합니다.	제어 센터에서 **테이블** 아이콘을 마우스의 오른쪽 버튼으로 클릭하고 **작성 —> 마법사를 사용한 테이블**을 선택하십시오.
테이블 공간 작성	새로운 테이블 공간을 작성합니다.	제어 센터에서 **테이블 공간** 아이콘을 마우스의 오른쪽 버튼으로 선택하고 **작성 —> 마법사를 사용한 테이블 공간**을 선택하십시오.
색인 작성	사용자의 모든 조회를 작성하고 제거하기 위해 색인화합니다.	제어 센터에서 **색인** 아이콘을 마우스의 오른쪽 버튼으로 클릭하고 **작성 —> 마법사를 사용한 색인**을 선택하십시오.
성능 구성	업무 요구조건에 맞게 구성 매개변수를 갱신하여 데이터베이스의 성능을 조정합니다.	제어 센터에서 성능을 조정하려는 데이터베이스를 마우스의 오른쪽 버튼으로 클릭하고 **마법사를 사용한 성능 구성**을 선택하십시오. 파티션된 데이터베이스에 대해 데이터베이스 파티션 뷰로부터 성능을 조정하려는 첫번째 파티션을 마우스의 오른쪽 버튼으로 클릭하고 **마법사를 사용한 성능 구성**을 선택하십시오.
데이터베이스 복원	실패 후에 데이터베이스를 복구합니다. 사용할 백업 위치 및 재작동할 로그 기록을 이해하는 데 도움을 줍니다.	제어 센터에서 복원하려는 데이터베이스를 마우스의 오른쪽 버튼으로 클릭한 다음 **복원 —> 마법사를 사용한 데이터베이스**를 선택하십시오.

문서 서버 설정

기본적으로 DB2 정보는 지역 시스템에 설치됩니다. 이는 DB2 정보에 액세스해야 하는 모든 사람이 동일한 파일을 설치해야 함을 의미합니다. DB2 정보를 한 위치에 저장하려면, 다음과 같이 하십시오.

1. 지역 시스템의 \sqllib\doc\html에 있는 모든 파일과 서브디렉토리를 웹 서버로 복사하십시오. 각 책은 책을 구성하는 데 필요한 모든 HTML 및 GIF 파일이 들어 있는 서브디렉토리를 가집니다. 디렉토리 구조가 변경되지 않게 하십시오.

2. 새로운 위치에 있는 파일을 찾도록 웹 서버를 구성하십시오. 보다 자세한 정보는 *설치 및 구성 보충 설명서*의 부록 NetQuestion을 참조하십시오.
3. Java 버전의 정보 센터를 이용하는 경우, 모든 HTML 파일에 대한 기본 URL을 지정할 수 있습니다. 책 목록에 대해서는 URL을 사용해야 합니다.
4. 책 파일을 열람할 수 있게 되면, 다음과 같이 자주 열람하는 주제 항목에 대해서는 북마크를 설정할 수 있습니다. 다음의 페이지들을 북마크로 설정해 두면 도움이 될 것입니다.
   - 책 목록
   - 자주 이용하는 책의 목차
   - ALTER TABLE 주제와 같은 자주 참조하는 항목
   - 검색 양식

DB2 Universal Database 온라인 문서 파일을 중앙 시스템에서 제공하는 방법에 대한 정보를 보려면 *설치 및 구성 보충 설명서*의 부록 NetQuestion을 참조하십시오.

정보 온라인 검색

HTML 파일에서 정보를 찾으려면, 다음 방법 중 하나를 사용하십시오.

- 맨 위 프레임에서 **검색**을 클릭하십시오. 특정 주제를 찾으려면 검색 형식을 사용하십시오. 이 기능은 Linux, PTX 또는 Silicon Graphics IRIX 환경에서는 사용할 수 없습니다.
- 맨 위 프레임에서 **색인**을 클릭하십시오. 책에서 특정 주제를 찾으려면 색인을 사용하십시오.
- 책에서 특정 주제를 찾으려면 목차나 도움말의 색인 또는 HTML 책을 표시하고 웹 브라우저의 찾기 기능을 사용하십시오.
- 특정 주제로 빨리 리턴하려면 웹 브라우저의 북마크 기능을 사용하십시오.
- 특정 주제를 찾으려면 정보 센터의 검색 기능을 사용하십시오. 922 페이지의 『정보 센터로 정보에 액세스』에서 자세한 내용을 참조하십시오.

부록G. 주의사항

IBM은 이 책에서 논의된 제품, 서비스 또는 기능을 다른 나라에서는 제공하지 않을 수 있습니다. 현재 사용자가 사용할 수 있는 제품 및 서비스에 대한 정보는 해당 지역의 IBM 영업대표에게 문의하십시오. IBM 제품, 프로그램 또는 서비스를 언급했다고 해서 반드시 IBM 제품, 프로그램 또는 서비스만을 사용해야 함을 의미하지는 않습니다. IBM의 지적 소유권을 침해하지 않는 기능상으로 동등한 타사의 제품, 프로그램 또는 서비스를 대신 사용할 수 있습니다. 그러나, 타사 제품, 프로그램 또는 서비스의 운영에 대한 평가 및 검증은 사용자의 책임입니다.

IBM은 이 책에서 다루고 있는 특정 내용에 대한 특허를 보유하고 있거나 출원중일 수 있습니다. 이 문서를 제공한다고 해서 그러한 특허에 대한 사용권까지 부여하는 것은 아닙니다. 특허 사용권에 대한 문의는 다음 주소로 하십시오.

135-270
서울특별시 강남구 도곡동 467-12, 군인공제회관빌딩
한국 아이.비.엠 주식회사
지적 재산권부

2바이트(DBCS) 정보에 관한 사용권 문의는 사용자 국가의 IBM 지적 재산권부나 다음 주소로 서면 문의하십시오.

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

다음 사항은 영국이나 이 조항이 현지법과 상충되는 나라에는 적용되지 않습니다. IBM에서는 이 책을 명시적 또는 암시적인 어떠한 종류의 보증없이『있는 그대로』제공하므로, 판매 가능성을 보장하거나 특정 목적에 적합한지 여부에 대해서는 책임질 수 없습니다. 일부 국가에서는 특정 거래의 명시적 또는 암시적인 보증을 부인하는 문장을 허용하지 않으므로, 이 사항이 사용자에게 적용되지 않을 수도 있습니다.

© Copyright IBM Corp. 1993, 2000

이 책에는 기술상 부정확한 내용이나 인쇄상의 오류가 있을 수 있습니다. 이 책의 내용은 정기적으로 변경되며, 이들 변경사항은 개정판에 통합됩니다. IBM은 사전 통지없이 언제든지 이 책에 설명된 제품과 프로그램을 개선 및 변경할 수 있습니다.

이 책에서 타사의 웹 사이트를 언급한 것은 단지 편의를 위해서일 뿐이며 이런 웹 사이트를 추천하려는 의도는 아닙니다. 이런 웹 사이트의 데이터가 이 IBM 제품에 대한 데이터의 일부는 아니므로 이런 웹 사이트 사용에 대한 책임은 사용자가 져야 합니다.

IBM은 독자가 제공한 정보를 적절한 방식으로 사용하거나 배포할 수 있으며, 제공한 독자는 이에 대해 책임을 지지 않습니다.

이 프로그램의 사용권자가 (i) 독립적으로 작성된 프로그램과 다른 프로그램(이 프로그램을 포함한) 사이의 정보 교환과 (2) 교환된 정보의 공동 사용을 목적으로 그 프로그램에 대한 정보를 원하는 경우, 다음 주소로 문의하십시오.

135-270
서울특별시 강남구 도곡동 467-12, 군인공제회관빌딩
한국 아이.비.엠 주식회사
소프트웨어 사업부

이러한 정보는 특정한 기간 및 조건하에 사용 가능하며, 어떤 경우에는 사용료를 지불해야 합니다.

이 책에 기술된 사용권 프로그램 및 이 프로그램에 사용가능한 모든 사용권 데이터는 IBM 고객 협약, IBM 국제 프로그래밍 사용권 협약 또는 이와 동등한 모든 협약 조건하에 IBM에서 제공됩니다.

여기에 제시된 어떠한 성능 데이터는 주위 환경에 따라 결정될 수 있습니다. 따라서, 다른 운영 체제에서 제시된 결과 값과 다를 수 있습니다. 몇몇 측정값은 개발 단계에서 얻은 값일 수 있습니다. 따라서 일반적인 사용자 시스템에서 얻은 값과 다를 수 있습니다. 또한 몇몇 측정값은 보외법을 통해 측정된 값입니다. 실제 값과는 다를 수 있습니다. 이 책의 사용자는 사용자의 특정 환경에 맞게 적용가능한 데이터를 변경해야 합니다.

타사 제품과 관련된 정보는 해당 제품의 공급자, 공개 발표 또는 기타 공개적으로 사용가능한 소스에서 확보한 것입니다. IBM은 이들 제품을 검사하지 않았고 성능 상의 정확성, 호환성 또는 타사 제품과 관련된 기타 주장을 확인할 수 없습니다. 타사 제품의 성능에 관한 문제는 해당 제품의 공급자에게 제기되어야 합니다.

IBM이 제시하는 방향 또는 의도에 관한 어떠한 언급도 특별한 통지없이 변경될 수 있습니다.

이 정보는 일상적인 비지니스 처리에 사용되는 데이터와 보고서의 예가 들어 있을 수 있습니다. 보다 구체적으로 예를 나타내기 위해 특정 개인, 회사, 상표 또는 제품 이름이 언급되는 경우가 있습니다. 여기서 언급된 이름은 가상의 이름이며 실제 비지니스 업체가 사용하는 이름 및 주소와 유사하다면 우연인 것입니다.

사용권:

이 정보에는 여러 운영 체제에서 프로그래밍 소스 언어로 예제 응용프로그램이 들어 있을 수 있습니다. 사용자는 이들 예제 프로그램을 IBM에게 비용을 지급하지 않고 복사, 수정 및 분배할 수 있습니다. 이들 예제 프로그램은 모든 조건에서 철저하게 검사되지 않았습니다. 따라서, IBM은 이들 프로그램에 대해 어떠한 보증도 할 수 없습니다.

이들 예제 프로그램의 각각의 사본이나 특정 부분은 다음과 같은 사용권 주의사항을 포함해야 합니다.

© (사용자 회사 이름) (년도). 이 코드의 일부는 IBM Corp. 예제 프로그램에서 발췌된 것입니다. © Copyright IBM Corp. (년도 입력). All rights reserved.

등록 상표

별표(*)로 표시된 다음의 용어는 전세계에서 IBM의 상표입니다.

ACF/VTAM
AISPO
AIX
AIX/6000
AIXwindows
AnyNet
APPN
AS/400
BookManager
CICS
C Set++
C/370
DATABASE 2
DataHub
DataJoiner
DataPropagator
DataRefresher
DB2
DB2 Connect
DB2 Extenders
DB2 OLAP Server
DB2 Universal Database
Distributed Relational Database Architecture
DRDA
eNetwork
Extended Services
FFST
First Failure Support Technology
IBM
IMS
IMS/ESA
LAN DistanceMVS
MVS/ESA
MVS/XA
Net.Data
OS/2
OS/390
OS/400
PowerPC
QBIC
QMF
RACF
RISC System/6000
RS/6000
S/370
SP
SQL/DS
SQL/400
System/370
System/390
SystemView
VisualAge
VM/ESA
VSE/ESA
VTAM
WebExplorer
WIN-OS/2

다음 용어는 해당 회사의 상표 또는 등록 상표입니다.

Microsoft, Windows 및 Windows NT는 Microsoft Corporation의 상표 또는 등록 상표입니다.

Java 또는 모든 Java 관련 상표 및 로고 그리고 Solaris는 전세계에서 Sun Microsystems, Inc.의 상표입니다.

Tivoli 및 NetView는 전세계에서 Tivoli Systems Inc.의 상표입니다.

UNIX는 전세계에서 X/Open Company Limited가 독점권을 갖는 등록 상표입니다.

두 개의 별표(**)가 붙은 기타 회사 이름, 제품 이름 또는 서비스 이름은 해당 회사의 상표이거나 서비스 표시입니다.

색인

[가]

가변 길이 문자열 884
가중치, 정의 560
갱신 조작 539
검색
  복수 행 95
  온라인 정보 922, 925
  한 행 78
  행 108
결과 세트
  저장 프로시듀어에서 251
  SQL 프로시듀어로부터 리턴 276
결과 코드 18
경고 메시지, 절단 90
경보, 트리거에 의해 지원되는 538
계승
  ONLY절 제어 330
계층
  구조화 유형 317
고려사항
  데이터 값 제어 32
  데이터 관계 제어 34
  데이터로 액세스 28
  서버에서의 응용프로그램 논리 36
  DB2 응용프로그램 설계 26
고유 키 위반
  버퍼 삽입 628
고장 진단 및 해결
  저장 프로시듀어 260
  Visual Studio 사용 260
고정 또는 가변 길이 데이터 유형
  확장 UNIX 코드 고려사항 594
공유 메모리 크기에 대한 시스템 구성 매개변수 502
관찰 메소드 322
교착 상태
  다중 스레드 응용프로그램에서 610
  버퍼 삽입의 오류 628
  복수 문맥에서 방지 611
구문
  공백 문자 대체의 embedded SQL문 669
  문자 호스트 변수 674
  선언 절
    COBOL 769
    FORTRAN 795
  행 구분을 피하는 embedded SQL문 669
  C/C++에서 embedded SQL문 668
  C/C++용 embedded SQL문 주석 669
  C/C++의 그래픽 호스트 변수 676
  C/C++의 선언 섹션 671
  embedded SQL문
    COBOL 765
    COBOL의 주석 766
    FORTRAN 793
    FORTRAN의 주석 793
  REXX에서 embedded SQL문 주석 810
  REXX에서 SQL문 처리 808
구별 유형 412
  구별 유형 정의 304
  규정되지 않은 구별 유형 분석 305
  명백한 유형 지정 307
  조작
    예 307
  테이블 정의 306
구별 유형 관련된 포함의 예 311
구별 유형과 관련된 비교의 예 308
구별 유형과 관련된 지정 예 312
구별 유형에서 사용자 정의 전래 함수 예 311
구성 매개변수
  LOCKTIMEOUT 612
구성원 연산자, C/C++ 제한사항 693
구성자 함수 321
구조화 유형
  개요 316
  계승 316
  계층 317
  관찰 메소드 322
  구성자 함수 321
  구조화 유형 속성 정의 351
  동작 정의 322
  동적 유형 329
  메소드 호출 322, 351
  변환 메소드 321
  변환 함수와 함께 부속 유형 바인드인 377
  부속 유형 속성 검색 354
  속성 316
  속성 값 검색 322
  에 대한 정보 리턴 355
  유형 계층에서 부속 유형 액세스 328
  유형화 테이블 작성 324
  의 내부 ID 검색 344
  의 속성 갱신 321, 353, 355
  의 스키마 이름 검색 344
  의 유형 이름 검색 344
  이점 316
  인스턴스 비교 345
  인스턴스 작성 321
  인스턴스를 단일 값으로 검색 353

© Copyright IBM Corp. 1993, 2000

구조화 유형 *(계속)*
인스턴스화 가능 유형 329
인스턴스화 불가능 유형 329
저장 317
정적 유형 329
컬럼에 인스턴스 삽입 351
컬럼에 저장 349
컬럼에서 오브젝트 저장 327
클라이언트 응용프로그램으로 인스턴스 전달 367
표시 유형 320
하위 오브젝트 참조 319
행으로 저장 324
호스트 변수 선언 381
DESCRIBE문 382
MODE DB2SQL절 316
국가 코드
SQLCA의 SQLERRMC 필드 886
규정자 속성
상이한 플랫폼 887
패키지 887
규정화되지 않은 이름
해결 67
규정화되지 않은 참조 413
규정화되지 않은 함수 참조 예 427
규정화된 함수 참조 사용 예 426
그래픽 데이터
대만어 코드 세트 581, 585
일본어 코드 세트 581, 585
그래픽 데이터 유형
선택 695
그래픽 문자열
문자 변환 577
그래픽 상수
대만어 코드 세트 585
일본어 코드 세트 585
그래픽 오브젝트
Java에 대한 고려사항 750
그래픽 호스트 변수
COBOL 772
그래픽 호스트 변수 *(계속)*
C/C++ 676
기본 키 890
기호 대체, C/C++ 언어 제한사항 684
긴 필드 884

[나]

내재된 연결 885
널 종료기 707
널로 종료되는 문자 양식 C/C++ 유형 702
널(NULL) 값
수신, 준비 89

[다]

다중 사이트 갱신
개요 598
구성 매개변수 604
다중 사이트 갱신 응용프로그램을 위한 SQL 코딩 599
사용 시기 599
일반 고려사항 597
저장 프로시듀어에서의 고려사항 249
제한사항 604
지원 897
DB2 Connect 지원 897
다중 사이트 갱신 구성 마법사 923
다중 사이트 갱신 구성 매개변수
LOCKTIMEOUT 604
RESYNC_INTERVAL 604
SPM_LOG_NAME 605
SPM_NAME 604
SPM_RESYNC_AGENT_LIMIT 604
TM_DATABASE 604
TP_MON_NAME 604
다중 스레드
문맥 간의 교착 상태 방지 611
문맥 간의 데이터베이스 종속성 610
문맥 간의 응용프로그램 종속성 610
다중 스레드 *(계속)*
지침 608
DB2 응용프로그램에서 사용 607
닫힌 상태, 버퍼 삽입 628
대리 함수(surrogate function) 538
대만어
확장 UNIX 코드 고려사항 584
2바이트 고려사항 584
대만어 코드 세트 581
사용하여 응용프로그램 개발 584
C/C++ 고려사항 700
대만어 EUC 코드 세트
REXX 고려사항 826
대체 가능성 325, 328
대형 오브젝트(LOB)
값에 대한 프로그래밍 옵션 386
대형 오브젝트(LOB) 값 383
대형 오브젝트(LOB) 설명자 383
및 DB2 오브젝트 확장 297
위치 지정자 383, 386
사용 예 388, 394
표시기 변수 401
프로그래밍 시나리오 394
저장 297
조작 297
트리거, UDT 및 UDF와의 시너지 553
파일 참조 변수 383
사용 예 403
입력 값 401
출력 값 402
SQL_FILE_APPEND, 출력 값 옵션 402
SQL_FILE_CREATE, 출력 값 옵션 402
SQL_FILE_OVERWRITE, 출력 값 옵션 402
SQL_FILE_READ, 입력 값 옵션 402
Java에 대한 고려사항 750

대형 오브젝트(LOB) 값 383
대형 오브젝트(LOB) 데이터에 액세스하기 위한 제어 정보 384
대형 오브젝트(LOB) 설명자 383
대형 오브젝트(LOB) 저장 297
대형 오브젝트(LOB) 조작 297
대형 오브젝트(LOB) 컬럼의 최대 *크기*, 정의 384
대형 오브젝트(LOB)에서의 조작 규칙 297
대형 오브젝트(LOB)의 제한조건 메카니즘 297
데이터
  대용량 추출 630
  추출시 병목 방지 630
데이터 값 제어 고려사항
  데이터 유형 32
  응용프로그램 논리 및 변수 유형 34
  점검 옵션 보기 33
  점검 제한조건 33
  참조 무결성 고려사항 33
데이터 검색
  갱신 121
  두 번째 119
  뒤로 이동, 기술 118
데이터 관계 고려사항
  응용프로그램 논리 35
  참조 무결성 34
  트리거 35
데이터 구조
  사용자 정의, 다중 스레드 사용 608
  저장 프로시듀어에 대한 할당 214
  DB2DARI 저장 프로시듀어 조작 859
  SQLEDBDESC 565
데이터 구조의 직렬화 608
데이터 구조, 선언 13
데이터 복원 23
데이터 비일관성 21
데이터 유형
  그래픽 유형 선택 695
  데이터 값 제어 고려사항 32
  문자 변환 오버플로우 595
  변환
    DB2 및 COBOL간 781
    DB2 및 FORTRAN간 801
    DB2와 OLE DB 테이블 함수 간의 485
  변환 고려사항 370
  설명 14
  숫자 884
  오브젝트 지향 297
  유형 목록 및 UDF에서의 표시 453
  지원됨
    COBOL 780, 781
    COBOL, 규칙 784
    FORTRAN 800
    FORTRAN, 규칙 802
  클래스 데이터 구성원, C/C++;에서 선언 692
  포인터, C++에서 선언 690
  확장 UNIX 코드 고려사항 594
  10진
    FORTRAN 802
  BLOB 383
  CLOB 383
  COBOL 780
  C/C++ 701, 702, 707
  C/C++의 701
  C/C++의 2진 데이터용 708
  C/C++의 CLOB 708
  C/C++의 VARCHAR 708
  DB2 및 C/C++간의 변환 702
  DB2 및 REXX간 변환 817
  DB2와 OLE 자동화 간의 변환 475
  DBCLOB 383
  FOR BIT DATA, COBOL 784
  FORTRAN 800
  Java 712
데이터 유형 *(계속)*
  Java 저장 프로시듀어 (DB2GENERAL) 863
  OLE 자동화 475
  SQL 컬럼 유형, 목록 91
  UDF로 전달되는 방식 453
데이터 유형 맵핑 647
  기본값 648
  데이터 소스 작성 649
  특정 컬럼에 대해 작성 649
  ALTER NICKNAME문 650
  CREATE TYPE MAPPING문 649
데이터 일관성 21
데이터 전송
  갱신 121
데이터로의 액세스 고려사항
  조회 제품 사용 31
  DB2 Call Level Interface (DB2 CLI) 30
  embedded SQL 28
  JDBC 30
  Microsoft 스펙 31
  ODBC 30
  Perl 사용 31
  REXX 29
데이터베이스 관리 프로그램 API
  저장 프로시듀어를 사용한 호출 210
  정의, 샘플 프로그램 122
데이터베이스 백업 마법사 923
데이터베이스 설명자 블록 (SQLEDBDESC), 조합 순서 지정 565
데이터베이스 액세스
  다른 문맥 사용 607
  다중 스레드 사용 607
데이터베이스 작성 마법사 923
데이터베이스 작성, 조합 순서 지정 565
데이터베이스 추가 마법사 923, 924
독립 오브젝트로서 엔터티 모델링 297
동기점 관리 프로그램 605

동일하지 않은 코드 페이지 587
저장영역 할당 587
동일하지 않은 코드 페이지에 대한 메모리 할당 587
동일하지 않은 코드 페이지에 대한 저장영역 할당 587
동적 메모리, UDF에서 할당 499
동적 명령문
바인딩 66
동적 유형 329
동적 커서 887
동적 SQL
개요 145
고려사항 147
구문, 정적과의 차이점 146
구조화 유형에 대한 변환 그룹 359
권한 부여 고려사항 42
동적 SQL과의 비교 76
의 매개변수 표시문자 180
임의 명령문, 처리 172
정적 SQL과 비교 147
제한사항 146
지원되는 명령문, 목록 146
커서 처리 151
커서 처리, 샘플 프로그램 152
DB2 Connect 지원 881
EXECUTE IMMEDIATE문, 의 요약 146
EXECUTE문, 의 요약 146
PREPARE문, 의 요약 146
PREPARE, DESCRIBE 및 FETCH 사용 150
SQLDA 선언 162
동적 SQL에서의 지정 예 312
등록
OLE 자동 UDF 472
UDF 416
디버깅
저장 프로시듀어 249, 260
디버깅 *(계속)*
Stored Procedure Builder 사용 743
Java 프로그램 715
SQL 프로시듀어 279, 282
SQLJ 프로그램 715
Visual Studio 사용 260

[라]

래치
다중 스레드의 상태 607
런타임 서비스
다중 스레드, 래치에 미치는 영향 607
로케일
응용프로그램에서 구하기 567
DB2가 구하는 방법 567
루트 유형 318
루핑 응용프로그램
진단 639
리바인딩
설명 72
REBIND PACKAGE 명령 72
리턴 코드 18
SQLCA 구조 133
릴리스 정보 916
링크
개요 64

[마]

마법사
다중 사이트 갱신 구성 923
데이터베이스 백업 923
데이터베이스 복원 924
데이터베이스 작성 923
데이터베이스 추가 923, 924
색인 924
성능 구성 924
마법사 *(계속)*
타스크 완료 923
테이블 공간 작성 924
테이블 작성 924
매개변수 표시문자 189
동적 SQL 사용 180
임의 명령문 처리 172
프로그래밍 예 181
함수 예에서 426
SQLVAR 항목 181
메모리
LOB 위치 지정자를 사용하여 요구사항 감소 493
메모리 크기, UDF의 공유 502
메모리, UDF에서 동적 메모리 할당 499
메소드
구현 410
등록 416
연산자 호출 322
이론적 근거 410
작성 416, 433
정의 409
호출 322
메소드 호출
OLE 자동 UDF 473
메시지 파일, 정의 62
명령
EXCSQLSTT 898
FORCE 886
명령문
연결 885
호출 893
ACQUIRE 898
BEGIN DECLARE SECTION 14
COMMIT 22
COMMIT WORK RELEASE 899
CONNECT 20
CREATE STORGROUP 883
CREATE TABLESPACE 883

명령문 *(계속)*
DECLARE 898, 899
DELETE 883
DESCRIBE 898, 899
END DECLARE SECTION 14
GRANT 885
INCLUDE SQLCA 17
INSERT 883, 884
LABEL ON 898
PREPARE 899
ROLLBACK 23, 886
SELECT 883
SET CURRENT 899
UPDATE 883
명령문 핸들 189
명령행 처리기 821
유틸리티 프로토타입 50
명백한 유형 지정 및 구별 유형 307
명확한 커서 887
모음 찾기, UDF의 예에서 CLOB 접기 510
목표 파티션
버퍼 삽입 없이 동작 626
문맥
간의 교착 상태 방지 611
간의 데이터베이스 종속성 610
간의 응용프로그램 종속성 610
다중 스레드 DB2 응용프로그램에서 설정 607
문서 서버 설정 924
문서를 파일로 추출 (테이블에서의 CLOB 요소) 예 403
문자 대형 오브젝트(CLOB) 383
문자 변환 884
데이터 유형을 초과한 문자열 길이 오버플로우 595
문자열 길이 오버플로우 594
문자열 변환 규칙 595
발생 시점 573
문자 변환 884 *(계속)*
사전 처리 컴파일 및 바인딩 중에 572
유니코드(UCS-2) 596
응용프로그램 실행시 572
자국어 지원(NLS) 573
저장 프로시듀어 코딩 571, 596
지원되는 코드 페이지 575
프로그래밍 고려사항 568
확장 576
SQL문 코딩 569
문자 비교, 개요 561
문자 집합
확장 UNIX 코드(EUC) 578
문자 호스트 변수
COBOL 770
C/C++ 고정 길이 및 널로 종료 673
C/C++ 변수 길이 674
C/C++에서 고정 길이 및 널로 종료 673
C/C++에서의 변수 길이 674
FORTRAN 796
문자열 검색 및 정의 UDF 예 418
문제점 해결
숫자 변환 오버플로우 892
저장 프로시듀어 260

[바]

바인드 옵션
EXPLSNAP 67
FUNCPATH 67
INSERT BUF 627
QUERYOPT 67
바인드 파일
사전 처리 컴파일 옵션 61
역 호환성 67
REXX 응용프로그램에 대한 지원 820
바인딩
개요 65
고려사항 67
동적 명령문 66
바인드 파일 설명 유틸리티, db2bfd 69
옵션 65
지연 69
반복 읽기(RR), 기술 118
버퍼 삽입
개요 623
고유 키 위반 628
교착 상태 오류 628
닫는 명령문 624
닫힌 상태 628
버퍼 삽입에서의 오류 보고 628
버퍼 *크기* 624
비동기 특성 627
사용시 고려사항 627
사용시 제한사항 630
열린 상태 628
오류 탐지 627
이점 627
일부가 참 624
저장점 고려사항 203, 624
트랜잭션 로그 고려사항 627
행 그룹 627
CLP에서 지원되지 않음 630
INSERT BUF 바인드 옵션 627
long 필드 제한사항 630
SELECT 버퍼 삽입 628
버퍼 삽입 닫기 624
버퍼 삽입에서의 오류 탐지 627
버퍼 삽입의 버퍼 *크기* 624
버퍼 삽입의 비동기 특성 627
범위 지정된 참조
참조 무결성에 대한 비교 337
변경사항 구간 복원 23
변경사항 보관 23
변경사항 확약, 테이블 22

변수
SQLCODE 708, 785, 803
SQLSTATE 708, 785, 803
변수, REXX에서 사전 정의 811
변수, 선언 13
변환
문자 884
변환 그룹
동적 SQL 359
명명 권장사항 356
외부 루틴에 대해 358
정적 SQL 359
변환 메소드 321
변환 함수
구조화 유형 속성 맵핑 359
구조화 유형에 연관 355
부속 유형 매개변수 처리 372
부속 유형 바인드인 377
오브젝트를 외부 루틴으로 전달 359, 360
요약 테이블 371
클라이언트 응용프로그램으로 구조화 유형 전달 367
별명
고려사항, 제한사항 643
관련 정보 카탈로그화 643
뷰에 사용 646
CREATE NICKNAME문 644
보기
데이터 값 제어 고려사항 33
시스템 카탈로그 891
온라인 정보 921
복수 바이트 고려사항
일본어 및 대만어 EUC 코드 세트
COBOL 785
FORTRAN 803
C/C++에서 대만어 코드 세트 700
C/C++에서 일본어 코드 세트 700
REXX에서 대만어 EUC 코드 세트 826
복수 바이트 고려사항 *(계속)*
REXX에서 일본어 EUC 코드 세트 826
복수 바이트 코드 페이지
대만어 코드 세트 581, 584
일본어 코드 세트 581, 584
복수 트리거, 정렬 551
복원 마법사 924
복합 SQL
NOT ATOMIC 896
부속 유형 317
변환 함수 작성 372
변환 함수와 함께 바인드인 377
OUTER를 사용하여 속성 리턴 346
분리 레벨 647, 892
분리되지 않은 저장 프로시듀어 249
분산 서브섹션 (DSS) 622
분산 요청(DR)
최적화 652
코딩 650
분산 환경 881
불명확한 커서 887
뷰
데이터 소스 뷰 642
변경 341
시스템 카탈로그에 대한 제거 포함 341
제거 341
제한사항 341
뷰 변경 341
뷰 제거 341
블로킹 886
비동기 이벤트 607
비워진 버퍼 삽입 624
비일관성 상태 21

[사]

사용
Java UDF 466
Java 저장 프로시듀어 740
사용자 갱신가능 카탈로그 통계 유틸리티 프로토타입 51
사용자 정의 유형
DB2 Connect에서 지원 885
사용자 정의 유형 제거 340
사용자 정의 유형(UDT)
제한사항 제거 340
사용자 정의 조합 순서 889, 901
사용자 정의 함수(UDF), 응용프로그램 논리 고려사항 36
사전 처리 컴파일 63
개요 60
동적 SQL문 지원 146
복수 서버 액세스 63
예 60
옵션, 갱신가능 커서 108
플래거 유틸리티 63
DB2 Connect을 통해 호스트 또는 AS/400 응용프로그램 서버 액세스 63
사전 처리 컴파일 옵션
WCHARTYPE
NOCONVERT 250
사전 처리 컴파일러
개요 56
옵션 61
지원되는 언어 13
출력의 유형 61
COBOL 761
C/C++ trigraph 절차 662
C/C++ 기호 대체 661
C/C++ 매크로 처리 661
C/C++ 문자 세트 662
C/C++ 언어 693
C/C++ 언어 디버깅 668
C/C++ #include macro 661
DB2 Connect 지원 883, 886
FORTRAN 787
산술 오류
UDF에서 438

삽입
버퍼 삽입 없이 626
상위 유형 317
상이한 DB2 제품 사이의 차이점 882
색인 마법사 924
색인 확장 299
샘플 프로그램
상호 플랫폼 915
위치 835
API(Application Program Interface) 835
embedded SQL문 835
HTML 915
Java UDF 466
Java 저장 프로시듀어 740
서로 다른 구별 유형 예가 관련된 지정 313
서버 옵션 652
서브테이블
서브테이블에서 속성 계승 330
작성 325
선언
표시기 변수 89
호스트 변수, 규칙 84
선언 절
명령문에 대한 규칙 84
COBOL 769, 783
C/C++의 671, 705
db2dclgn으로 작성 86
FORTRAN 795, 802
선언된 임시 테이블 197
선행 처리기 기능과 SQL 사전 처리 컴파일러 684
설명자 핸들 189
설치
Netscape 브라우저 921
성공 코드 18
성능
대형 오브젝트(LOB) 385
데이터 블록 전달 619
성능 *(계속)*
동적 SQL 캐싱 76
방향이 지정된 DSS 622
버퍼 삽입 향상 623
영향 요인, 정적 SQL 76
읽기 전용 커서로 개선 108
정적 SQL문 사전 처리 컴파일 70
지역 바이패스 향상 623
파티션된 환경 향상 621
패키지 최적화 70
향상
저장 프로시듀어 사용 208
LOB 위치 지정자를 사용하여 증가 493
NOT FENCED 저장 프로시듀어 249
READ ONLY 커서로 개선 621
UDF 410
성능 구성 마법사 924
성능 및 구별 유형 303
성능상의 이점
버퍼 삽입을 사용하여 627
세마포어 610
소스 레벨 디버거 및 UDF 535
소스 파일
수정된 소스 파일, 정의 61
요구사항 62
작성, 개요 57
파일 이름 확장자 61
SQL 파일 확장 57
소유자 속성
패키지 887
속성 316
숫자 데이터 유형 884
숫자 변환 오버플로우 892
숫자 호스트 변수
COBOL 769
C/C++ 672
FORTRAN 795
스냅샷 모니터
일시정지했거나 루핑 응용프로그램 진단 639
스레드간 문맥 전달 608
스칼라 함수 414
call-type 인수의 내용 444
스크래치 패드 고려사항
OLE 자동 UDF 473
스크래치 패드 및 UDF 468, 488
스크래치 패드, UDF로 전달 435
스키마 이름 및 UDF 413
시스템 카탈로그
뷰 포함 제거 341
사용 891
시스템 카탈로그 뷰
유틸리티 프로토타입 51
식별 순서 560
식별 컬럼 196
신호 핸들러
설치, 샘플 프로그램 122
SQL문으로 136
신호, 두 개의 함수 및 동일 413
실행불가 SQL문
DECLARE CURSOR 20
INCLUDE 20
INCLUDE SQLDA 20
심각한 오류
파티션된 환경에서의 고려사항 637

[아]

안내
데이터 값 제어 32
데이터 관계 제어 34
데이터로 액세스 28
서버에서의 응용프로그램 논리 36
DB2 응용프로그램 설계 26
언어 식별자
책 915

연결
- 내재된 연결 885
- 널(NULL) CONNECT 885
- CONNECT RESET문 885
- CONNECT TO문 885

연결 해제
- CMS 응용프로그램 22
- DB2로의 23

연결 핸들 189
연쇄 890
연쇄 트리거 551
연합 시스템
- 데이터 무결성 647
- 데이터 소스 테이블, 뷰
  - 고려사항, 제한사항 643
  - 별명 642
  - 정보 카탈로그화 643
- 데이터 소스 함수 653
- 데이터 유형 맵핑 647
- 별명 642
- 분리 레벨 647
- 분산 요청(DR) 650
- 서버 옵션 652
- 소개 641
- 컬럼 옵션 645
- 함수 맵핑 653
- 함수 맵핑 옵션 655
- pass-through 656

연합 시스템의 데이터 소스
- 데이터 유형 맵핑 648
- 분리 레벨 맵핑 647
- 분산 요청(DR)을 사용하여 조회 650
- 테이블, 뷰 액세스 642
- 함수 호출 653
- DB2 함수 맵핑 653
- pass-through를 사용하여 조회 656

열린 상태, 버퍼 삽입 628
예
- 검색 및 갱신에서 매개변수 표시문자 사용 181

예 *(계속)*
- 구별 유형 관련 포함 311
- 구별 유형과 관련된 비교 308
- 구별 유형과 관련된 지정 312
- 구별 유형에서 사용자 정의 전래 함수 311
- 동적 SQL에서의 지정 312
- 문서를 파일로 추출 (테이블에서의 CLOB 요소) 403
- 서로 다른 구별 유형이 관련된 지정 313
- 지원되는 SQL 데이터 유형에 대한 샘플 SQL 선언 절 705
- ADHOC.SQC C 프로그램 목록 177
- BLOB 데이터 선언 680
- CLOB 값에 대해 작업할 위치 지정자 사용 388
- CLOB 데이터 선언 681
- CLOB 위치 지정자 682
- CLOB 컬럼에 데이터 삽입 408
- CLOB 파일 참조 683
- COBOL을 사용하여 BLOB 선언 774
- COBOL을 사용하여 BLOB 위치 지정자 선언 775
- COBOL을 사용하여 BLOB 파일 참조 선언 775
- COBOL을 사용하여 CLOB 선언 774
- COBOL을 사용하여 DBCLOB 선언 774
- CREATE DISTINCT TYPE 사용 비용 305
- CREATE DISTINCT TYPE 사용 재개 305
- CREATE TABLE을 사용한 응용프로그램 양식 306
- CREATE TABLE을 사용한 판매 306

예 *(계속)*
- DB2Appl.java 722
- DBCLOB 데이터 선언 681
- DYNAMIC.CMD REXX 프로그램 목록 160
- Dynamic.java Java 프로그램 목록 156
- DYNAMIC.SQB COBOL 프로그램 목록 158
- DYNAMIC.SQC C 프로그램 목록 154
- FORTRAN을 사용하여 BLOB 선언 798, 799
- FORTRAN을 사용하여 BLOB 파일 참조 선언 800
- FORTRAN을 사용하여 CLOB 파일 위치 지정자 선언 799
- FORTRAN의 문자 호스트 변수에 대한 구문 796, 797
- Java 애플릿 722
- LOB 표현식 평가 연기 394
- LOBEVAL.SQB COBOL 프로그램 목록 398
- LOBEVAL.SQC C 프로그램 목록 396
- LOBFILE.SQB COBOL 프로그램 목록 406
- LOBFILE.SQC C 프로그램 목록 404
- LOBLOC.SQB COBOL 프로그램 목록 391
- LOBLOC.SQC C 프로그램 목록 389
- REXX용 SQLEXEC, SQLDBS 및 SQLDB2 등록 806
- SQLEXEC, SQLDBS 및 SQLDB2 등록 807
- SQL문에서 클래스 데이터 구성원 사용 692
- UNION에서의 구별 유형 사용 314

예 *(계속)*
- V5SPCLI.SQC C 프로그램 목록 876
- V5SPSRV.SQC C 프로그램 목록 879
- Varinp.java Java 프로그램 목록 185
- VARINP.SQB COBOL 프로그램 목록 187
- VARINP.SQC C 프로그램 목록 183

오류 메시지
- 경고 조건 플래그 133
- 시간소인, 사전 처리 컴파일시 71
- 오류 조건 플래그 133
- 제외 조건 플래그 133
- SQLCA 구조 133
- SQLSTATE 134
- SQLWARN 구조 134

오류 보고 638

오류 조절
- C/C++ 언어 사전 처리 컴파일러 668
- C/C++용 include 파일 666

오류 처리
- 개요 133
- 루핑 응용프로그램에서 639
- 병합된 복수 SQLCA 구조 637
- 보고 638
- 사전 처리 컴파일시 62
- 오류를 리턴하는 파티션 식별 639
- 일시정지한 응용프로그램에서 639
- 재설정 19
- 파티션된 환경에서의 고려사항 636
- include 파일
  - COBOL 762, 763, 765
  - FORTRAN 789, 791
- SQLCA 구조 638
- SQLCA 이용 17
- SQLCODE 638

오류 처리 *(계속)*
- WHENEVER문 18
- WHENEVER문 사용 135

오류 코드 18

오버로드
- 저장 프로시듀어 이름 215
- 함수 이름 413

오브젝트 관계형
- 데이터 유형 297
- 응용프로그램 도메인 및 오브젝트 지향 297
- 정의 297
- 제한조건 메카니즘 297
- 트리거 297
- DB2 오브젝트 확장 사용 이유 297
- LOB 297
- UDT 및 UDF 297

오브젝트 링크 및 삽입(OLE) 471

오브젝트 식별자
- 에 대한 표현 유형 선택 334
- 자동으로 생성 347
- 제한조건 작성 348

오브젝트 식별자 컬럼 319, 320
- 명명 330

오브젝트 인스턴스
- OLE 자동 UDF 473

오브젝트 지향 데이터 유형의 인스턴스, 저장 297

오브젝트 지향 및 UDF 410

오브젝트 지향 확장 및 구별 유형 303

오브젝트 지향 COBOL 제한사항 786

온라인 도움말 919

온라인 정보
- 검색 925
- 보기 921

완료 코드 18

완전한 이름 482

외부 키 890

원격 데이터 오브젝트 지원
- DB2에서 지원 31

원격 작업 단위 597

원시 이동, DB2 점검 535

유니코드
- Java 750

유니코드(UCS-2)
- 대만어 코드 세트 581
- 문자 변환 596
- 문자 변환 오버플로우 594
- 일본어 코드 세트 581
- UDF 고려사항 585

유틸리티 테스트 및 디버깅
- 데이터베이스 시스템 모니터 49
- 시스템 카탈로그 뷰 49
- 시스템 카탈로그 통계 갱신 49
- 플래거 49
- Explain 기능 49

유틸리티 API
- COBOL 응용프로그램에 대한 include 파일 763, 765
- C/C++ 응용프로그램용 include 파일 666
- FORTRAN 응용프로그램에 대한 include 파일 792

유형
- ROWID 885

유형 데코레이션
- 저장 프로시듀어의 248

유형 또는 인수, UDF에서의 승격 453

유형 맵핑 647
- 제한사항 제거 340
- OLE 자동 유형 및 BASIC 유형 475
- OLE 자동 유형 및 C++ 유형 475

유형 장식
- UDF에서 504

유형 장식물 고려사항
- C++ 662

유형 참조
- 비교 320
- 유형변환 320

유형 컬럼
개요 349
작성 349
유형변환 412
SQL 유형 및 OLE 자동 유형 사이 474
UDF 430
유형화 뷰
루트 유형에서 작성 338
부속 유형에서 작성 338
의 본문 339
작성 338
참조 컬럼에 범위 지정 340
유형화 뷰 작성 338
유형화 테이블
간의 관계 정의 326, 335
계층 위치 결정 330
로 오브젝트 삽입 333
범위 정의 332
부속 유형 속성 리턴 346
서브테이블 작성 325
에서 데이터 선택 342
오브젝트 식별자 삽입 334
오브젝트 식별자 컬럼 330
유형 계층에서 부속 유형 액세스 325
자체 참조 336
작성 329
정의 324
특권 제어 331
융통성 및 구별 유형 303
응용프로그램 논리
데이터 값 제어 고려사항 34
데이터 관계 고려사항 35
응용프로그램 논리 고려사항
사용자 정의 함수 36
저장 프로시듀어 36
트리거 37
응용프로그램 도메인 및 오브젝트 지향 297
응용프로그램 설계
데이터 값 제어 고려사항 32
데이터 관계 34
데이터 두 번째 검색 119
데이터 전달, 안내 171
데이터로 액세스 28
데이터베이스 값 수신 89
동적 SQL 캐싱 76
동적 SQL, 의 개요 145
매개변수 표시문자 사용 180
문자 변환 고려사항 568
바인딩 57
변수없이 명령문 실행 146
사전 처리 컴파일 및 바인딩 57
샘플 프로그램 122
서버에서의 논리 36
안내 26
오류 처리, 안내 135
일반 사용자 요청 저장 174
입력-SQLDA 저장 프로시듀어, 샘플 878
입력-SQLDA 프로시듀어, 샘플 873
저장 프로시듀어에서 문자 변환 571
정적 SQL, 사용시 이점 76
충분한 SQLVAR 엔터티 선언 162
커서 처리, 고려사항 97
테이블 함수 고려사항 490
특수 문자에 대한 코드 포인트 570
패키지 이름 바꾸기 65
2바이트 문자 지원(DBCS) 570
COBOL 요구사항, include 파일 762
COBOL 일본어 및 대만어 EUC 고려사항 785
DB2 응용프로그램 코딩, 개요 12
OLE 자동 UDF 471
REXX 요구사항, 루틴 등록 806
SELECT문 설명 166
SQLDA 구조 작성, 안내 167
SQL문의 문자 변환 569
응용프로그램 설계 *(계속)*
UDF에서 LOB 위치 지정자 사용 493
varying-list 명령문, 처리 173
응용프로그램 이식 881
응용프로그램 환경, 프로그래밍 12
이 책에서 사용되는 규약 9
이 책의 사용법 5
이름 바꾸기, 패키지 65
이식성 191
인수 유형, UDF에서의 승격 453
인수, DB2에서 UDF로 전달 435
인스턴스화 가능성 329
인터랙티브 SQL
처리, 샘플 프로그램 174
인터럽트, SIGUSR1 136
일관성
데이터 21
일관성 있는 조작 및 구별 유형 303
일반 호출, UDF에 대한 443
일본어 및 대만어 EUC 코드 세트
COBOL 고려사항 785
FORTRAN 고려사항 803
일본어 코드 세트 581
사용하여 응용프로그램 개발 584
C/C++ 고려사항 700
일본어 EUC 코드 세트
REXX 고려사항 826
일시정지한 응용프로그램
진단 639
임시 테이블 197
입력 데이터를 검증하기 위한 SIGNAL SQLSTATE SQL문 538
입력 및 출력 파일
COBOL 761
C/C++ 663
FORTRAN 788
입력 파일 확장자, C/C++ 언어 663

[자]

자국어 지원(NLS)
  고려사항 560
  문자 변환 573
  코드 페이지 573
  혼합 바이트 데이터 884
자동 서버, OLE 472
자체 참조 유형화 테이블 336
자체 참조 테이블 890
작성
  Java UDF 466
  Java 저장 프로시듀어 740
  OLE 자동 UDF 472
작성가능한 단일 사용 OLE 자동 서버 478
작성가능한 복수 사용 OLE 자동 서버 478
작성자 속성
  패키지 887
작업 단위
  분산 597
  완료 97
  원격 597
  커서 고려사항 97
작업 환경
  설정 45
  테스트 데이터베이스, 작성 안내 45
잠금
  버퍼 삽입 오류 628
  시간종료 890
  페이지 레벨 890
  행 레벨 890
잠금시 시간종료 890
재사용 및 UDF 410
재입력
  UDF 488
저장 프로시듀어
  개요 207, 214
  결과 세트 리턴 251
저장 프로시듀어 *(계속)*
  경로 216
  그래픽 호스트 변수 고려사항 248
  다중 사이트 갱신 고려사항 249
  대만어 코드 세트 586
  디버깅 249
    Stored Procedure Builder 사용 743
  매개변수 215, 229, 230
  매개변수 모드 선언 215
  문자 변환 571
  문자 변환, EUC에 대한 596
  아키텍처 211
  예 229
  오버로드된 이름 215
  요구사항 211
  위치 216
  응용프로그램 고장 진단 및 해결 260
  응용프로그램 논리 고려사항 36
  이점 208
  일반 893
  일본어 코드 세트 586
  입력-SQLDA 저장 프로시듀어, 샘플 878
  입력-SQLDA 프로시듀어, 샘플 873
  저정영역 할당 214
  제한사항 233
  주 함수로서 작성 219
  중첩 232
  지원되는 언어 218
  코드 페이지 고려사항 247
  클라이언트 응용프로그램 214
  허용된 SQL문 231
  호스트 변수 213
  호출 214
  CALL문 214, 274
  CONTAINS SQL절 231
  CREATE PROCEDURE를 이용한 레지스터 214
저장 프로시듀어 *(계속)*
  CREATE PROCEDURE문 214
  C++ 고려사항 248
  db2dari 실행파일 233
  DBINFO 구조 전달 228
  EXTERNAL절 216
  FOR BIT DATA 고려사항 247
  Java 작성 및 사용 740
  Java로 등록 740
  LANGUAGE절 218
  NOT FENCED 249
  OUT 매개변수 클라이언트 프로그램 237
  PARAMETER STYLE절 221
  PROGRAM TYPE절 219
  REXX 변수 초기화 823
  REXX 응용프로그램 823
  SQLDA 및 SQLCA 구조 사용 861
저장 프로시듀어(DB2DARI)
  데이터 구조 사용 860
  리턴 값 862
  매개변수
    sqlda.n.SQLDAT 860
    sqlda.n.SQLDATALEN 860
    sqlda.n.SQLDATATYPE_NAME 860
    sqlda.n.SQLIND 860
    sqlda.n.SQLLEN 860
    sqlda.n.SQLLONGLEN 860
    sqlda.n.SQLNAME.data 860
    sqlda.n.SQLNAME.length 860
    sqlda.n.SQLTYPE 860
    sqlda.SQLDABC 860
    sqlda.SQLDAID 860
    sqlda.SQLN 860
  제한사항 859
  표시기 변수 사용 859
  호출 변환
    매개변수 변환 859
    SQL_API_FN 859

저장 프로시듀어(DB2DARI) *(계속)*
NOT FENCED 872
저장된 오브젝트의 의미론 조작 297
저장영역
충분한 SQLVAR 엔터티 선언 162
행 보유를 위한 할당 166
저장점 200
버퍼 삽입 203
블로킹 커서 203
제한사항 201
중첩 201
최소단위 복합 SQL 201
트리거 201
Data Definition Language 202
SET INTEGRITY문 202
XA 트랜잭션 관리 프로그램 204
저장점, 버퍼 삽입 고려사항 624
전래 UDF 310
전이 변수, OLD와 NEW
트리거 이벤트의 유형을 기반으로 한 전이 변수 544
전이 테이블
트리거 이벤트 유형에 기초하여 546
OLD_TABLE 및 NEW_TABLE 546
전제조건, 프로그래밍 12
절 번호 898
절단
표시기 변수 90
호스트 변수 90
정렬 순서
정의 889
조합 순서 889
정렬, 조합 순서 지정 565
정보 센터 922
정적 유형 329
정적 SQL
개요 76
고려사항 147
구조화 유형에 대한 변환 그룹 359
정적 SQL *(계속)*
데이터 검색 및 조작용 명령문 코딩 83
동적 SQL과 비교 147
사전 처리 컴파일, 이점 70
샘플 프로그램 77
정적 갱신 프로그래밍 예 122
호스트 변수 사용 84
DB2 Connect 지원 881
제한사항
버퍼 삽입 사용시 630
저장 프로시듀어(DB2DARI) 859
COBOL 761
C/C++의 684
FORTRAN 788
REXX 806
UDF에 대한 502
조건 핸들러
개요 268
예 270
SQL 프로시듀어 268
조정자 노드
버퍼 삽입 없이 동작 626
조합 순서
개요 560
대소문자 독립 비교 562
문자 비교에 사용 561
복수 바이트 문자 561
샘플 566
식별 순서 560
정렬 순서 예 564
지정 565
코드 포인트 560
C/C++에서 include 파일 665
EBCDIC 2진 조합 시뮬레이트 901
EBCDIC 및 ASCII 889
EBCDIC 및 ASCII 정렬 순서 예 564
include 파일
COBOL 763
조합 순서 *(계속)*
include 파일 *(계속)*
FORTRAN 790
조합 정렬
대만어 코드 세트 586
일본어 코드 세트 586
조회 제품, 데이터로의 액세스 고려사항 31
주석
SQL, 규칙 766, 793
주석, SQL, 규칙 669
중요한 절 610
중요한 절 루틴, 다중 스레드에서, 지침 610
중첩 저장 프로시듀어 232
결과 세트 리턴 274
매개변수 전달 274
반복 275
제한사항 275
SQL 프로시듀어 274
지수화 및 UDF 정의 예 417
지역
SQLCA의 SQLERRMC 필드 886
지역 바이패스 623

[차]

참조
를 사용한 관계 정의 326
참조 제한조건과의 비교 326
참조 해제 연산자 326
참조 무결성 890
데이터 관계 고려사항 34
범위 지정된 참조에 대한 비교 337
참조 무결성 제한조건
데이터 값 제어 고려사항 33
참조 범위 332
참조 유형
비교 335
에 대한 표현 유형 선택 334
정의 319

참조 컬럼
  유형화 뷰에 범위 지정 340
참조 해제 연산자 326
  를 사용하는 조회 342
참조된 컬럼
  범위 정의 332
책 907, 917
총계 함수 414
최소단위 복합 SQL
  DB2 Connect에서 지원 안함 896
최소단위 복합 SQL이 아님 896
최신 정보 916
최종 호출, UDF에 대한 443
최초 호출, UDF에 대한 443
추출
  대용량의 데이터 630
출력 파일 확장자, C/C++ 언어 663

[카]

카운팅 및 UDF 정의 예 421
캡슐화 및 구별 유형 303
커서
  갱신가능 108
  다중 행 검색 95, 96
  동적 887
  명명, REXX 809
  명확한 108
  명확함 887
  불명확한 887
  샘플 프로그램 109
  선언 97
  이름과 정의 96
  읽기 전용 96, 108
  읽기 전용, 응용프로그램 요건 97
  작업 단위(UOW) 완료 97
  처리, 동적 SQL 151
  처리, 샘플 프로그램 99, 152
  처리, 요약 96
  테이블 끝에 두기 121
  CLI에서 사용 189
커서 *(계속)*
  COMMIT문 발행 97
  FOR FETCH ONLY 108
  SQLDA 구조의 처리 167
  WITH HOLD 선언 97
커서 안정성(CS) 890
컬럼
  널 값 설정 90
  널(NULL) 입력 가능 데이터 컬럼에서의 표시기 변수 사용 94
  생성 196
  식별 196
  지원된 SQL 데이터 유형 91
  파생된 196
컬럼 생성 196
컬럼 옵션 332
  설명 645
  ALTER NICKNAME문 646
컬럼 유형, COBOL 작성 780
컬럼 유형, C/C++ 작성 701
컬럼 유형, FORTRAN 작성 800
컬럼 함수 414
컴파일 64
컴파일된 응용프로그램용 패키지 작성 60
컴파일된 응용프로그램, 패키지 작성 60
코드 세트
  SQLCA의 SQLERRMC 필드 886
코드 페이지
  동일하지 않은 상황 577, 587
  동일하지 않은 상황에서의 저장영역 할당 587
  로케일
    응용프로그램에서 구하기 567
    DB2가 로케일을 구하는 방법 567
  문자 변환 573
  바인딩 고려사항 67
  서버에서의 확장 처리 590
  응용프로그램에서의 확장 처리 590
코드 페이지 *(계속)*
  자국어 지원(NLS) 573
  저장 프로시듀어 고려사항 247
  지원되는 Windows 코드 페이지 566
  DB2CODEPAGE 레지스트리 변수 566
  SQLCA의 SQLERRMC 필드 886
코드 포인트 560
코드 포인트, 정의 560
콜 레벨 인터페이스(CLI)
  사용의 이점 191, 193
  embedded SQL 및 DB2 CLI 비교 189
콜렉션 888
콜렉션 ID 속성
  패키지 887
  AS/400용 DB2 Universal Database 888
클라이언트 변환
  개요 367
  데이터 유형변환 고려사항 370
  외부 UDF를 사용한 구현 369
  클라이언트 응용프로그램에서 인스턴스 바인드인 369
클라이언트 응용프로그램
  저장 프로시듀어 수행 214
클라이언트에 기초한 매개변수 검증
  확장 UNIX 코드 고려사항 591
클라이언트/서버
  코드 페이지 변환 573
클래스
  데이터 구성원, C/C++의 호스트 변수 692
키
  기본 890
  외부 890

[타]

테스트 데이터
 생성 47
테스트 데이터베이스
 작성 45
 추천사항 46
 CREATE DATABASE API 46
테스트 환경
 설정 45
 테스트 데이터베이스, 작성 안내 45
 파티션된 환경에서 636
테이블
 끝에 커서 두기 121
 데이터 소스 테이블 642
 변경사항 확약 22
 임시 197
 CREATE TABLE문의
  lob-options-clause 385
 CREATE TABLE문의
  tablespace-options-clause 385
테이블 공간 작성 마법사 924
테이블 이름
 규정화되지 않은 분석 67
테이블 작성 마법사 924
테이블 점검 제한조건, 데이터 값 제어 고려사항 33
테이블 함수 414, 436
 응용프로그램 설계 고려사항 490
 call-type 인수의 내용 445
 Java 466
 OLE DB 479
 SQL-result 인수 436
테이블 함수 예 422
테이블 함수에서 CARDINALITY 스펙 490
트랜잭션
 내재적 종료 24
 설명 21
 완료 작업 구간 복원 23
트랜잭션 *(계속)*
 완료 작업 확약 22
 저장점 200
 트랜잭션 시작 22
 트랜잭션 종료 22
 프로그램 종료
  COMMIT 및
   ROLLBACK문 23
트랜잭션 내재적 종료 24
트랜잭션 로그 고려사항
 버퍼 삽입 627
트랜잭션 로그, 버퍼 삽입 고려사항 627
트랜잭션 시작 22
트랜잭션 종료 22
트랜잭션 처리 모니터
 X/Open XA 인터페이스 615
트리거
 개요 539
 데이터 관계 고려사항 35
 및 DB2 오브젝트 확장 297
 복수 트리거, 정렬 551
 연쇄 551
 영향을 받는 행 집합 541
 응용프로그램 논리 고려사항 37
 이점 539
 전이 변수 544
 전이 테이블 546
 정의 537
 참조 제한조건과의 상호작용 551
 참조 제한조건, 트리거를 사용한 상호작용 551
 트리거 사용 이유 537
 트리거 수준 542
 트리거 이벤트 540
 트리거 조치 조건 548
 트리거 SQL문 548
 활성화 시간 543
 AFTER 트리거 543, 549
 BEFORE 트리거 543, 549
 DELETE 조작 540
트리거 *(계속)*
 INSERT 조작 540
 SQL 트리거문을 사용하는 함수
  RAISE_ERROR 내장 함수 549
 UDT, UDF 및 LOB와의 시너지 553
 UPDATE 조작 540
 WHEN절 548
트리거를 사용한 더욱 신속한 응용프로그램 개발 539
트리거를 사용한 더욱 용이한 유지보수 539
트리거를 사용한 비지니스 규칙의 전역 적용 539
특수 레지스터
 CURRENT EXPLAIN MODE 66
 CURRENT FUNCTION PATH 66
 CURRENT QUERY
  OPTIMIZATION 66

[파]

파생된 컬럼 196
파일 참조 변수
 사용 예 403
 입력 값 401
 출력 값 402
 LOB 조작용 383
파일 참조 변수를 사용한 대형 오브젝트 (LOB) 이동 383
파일 확장자
 샘플 프로그램 835
파티션된 환경
 성능 향상 621
 심각한 오류 고려사항 637
 오류 발생시 식별 639
 오류 처리시 고려사항 636
패키지
 속성 887
 시간소인 오류 71

패키지 *(계속)*
이름 바꾸기 65
작성 65
컴파일된 응용프로그램용 작성 60
REXX 응용프로그램에 대한 지원 820
패키지 속성
규정자 887
소유자 887
작성자 887
페이지 레벨 잠금 890
표시 유형 320
표시기 변수
널(NULL) 입력 가능 컬럼에서의 사용 94
선언 89
예 95
COBOL, 규칙 773
C/C++, 규칙 676
DB2DARI 저장 프로시듀어 사용 859
FORTRAN, 규칙 798
REXX, 규칙 811
표시기 변수 및 LOB 위치 지정자 401
표시기 테이블, COBOL 지원 778
프로그래밍 고려사항
다른 코드 페이지 간의 변환 560
복잡한 환경에서의 프로그래밍 560
자국어 지원 560
조합 순서 560
호스트 또는 AS/400 서버에 액세스 606
호스트 또는 AS/400 환경 881
COBOL 761
C/C++ 661
FORTRAN 787
REXX 805
X/Open XA 인터페이스 615
프로그래밍 프레임워크 25
프로그램 변수 유형, 데이터 값 제어 고려사항 34
플래거 유틸리티, 사전 처리 컴파일시 사용 63

[하]

함수
스칼라 함수 414
참조용 구문 424
총계 함수 414
컬럼 함수 414
테이블 함수 414
SQL 트리거문을 사용 549
함수 경로 및 UDF 413
함수 디렉토리 216
함수 맵핑
옵션 655
CREATE FUNCTION MAPPING문 653, 654
함수 변환
개요 360
외부 루틴에 매개변수 전달 365
SQL 본문 루틴으로 구현 363
함수 선택 알고리즘 및 UDF 413
함수 참조
UDF 요약 428
함수 참조 구문 424
함수 템플리트 654
함수 호출 예 425
핸들
명령문 핸들 189
설명자 핸들 189
연결 핸들 189
환경 핸들 189
핸들러
개요 268
예 270
행
검색 78
선택, SELECT INTO문 사용 78
행 *(계속)*
순서, 제어 120
커서로 다중 검색 108
테이블에서의 순서, 위치 121
트리거의 영향을 받는 행 집합 541
SQLDA를 사용한 검색 166
행 그룹
버퍼 삽입에서 627
행 레벨 잠금 890
행 블로킹
성능을 위한 사용자 정의 619
호스트 또는 AS/400
호스트 서버에 액세스 606
호스트 또는 AS/400 서버에서 데이터 확장 884
호스트 또는 AS/400 서버와 워크스테이션 사이의 차이점 898, 899
호스트 또는 AS/400 환경
프로그래밍 881
호스트 변수
구조화 유형 선언 381
그래픽 데이터 694
그래픽 데이터 유형 선택 695
그래픽 선언
COBOL 772
널로 종료되는 문자열, C/C++에서 조절 689
데이터 블록 전달에 사용 620
데이터 유형의 포인터로 선언 690
동적 SQL에서 사용 146
명명
COBOL 768
FORTRAN 795
복수 바이트 문자 엔코딩 694
선언 84
COBOL 769
FORTRAN 795
선언, 규칙 84
선언, 샘플 프로그램 122
선언, 예 87

호스트 변수 *(계속)*
저장 프로시듀어 고려사항 248
저장 프로시듀어 초기화 213
저장 프로시듀어에서 할당 213
정의 84
정적 SQL 84
참조
COBOL 768
FORTRAN 794
컬럼에 사용할 정의 방법 판별 17
클래스 데이터 구성원, C/C++에서 조절 692
COBOL에서 파일 참조 선언 775
COBOL에서 LOB 데이터 선언 773
COBOL에서 LOB 위치 지정자 선언 774
C/C++에서 참조 670
C/C++에서 초기화 684
C/C++에서 파일 참조 선언 683
C/C++에서 LOB LOB 위치 지정자 선언 682
C/C++에서 LOB 데이터 선언 679
C/C++에서의 그래픽 선언 676
C/C++에서의 명명 670
C/C++에서의 선언 671
db2dclgn을 사용하여 선언 86
FORTRAN에서 파일 참조 선언 800
FORTRAN에서 LOB 데이터 선언 798
FORTRAN에서 LOB 위치 지정자 선언 799
FORTRAN, 개요 794
REXX 810
REXX에서 명명 810
REXX에서 조회 810
REXX에서 파일 참조 선언 815
REXX에서 LOB 위치 지정자 선언 814

호스트 변수 *(계속)*
REXX에서 LOB 호스트 변수 지움 816
REXX의 LOB 데이터 813
SQL문에 연관시키기 16
SQL에서 참조, 예 88
variable list 명령문을 사용한 선언 173
WCHARTYPE 사전 처리 컴파일러 옵션 696
호출 규약
UDF에 대한 453
혼합 바이트 데이터 884
혼합 코드 세트 환경
응용프로그램 설계 586
혼합 확장 UNIX 코드 고려사항 583
화면과 키보드에 대한 입력과 출력 및 UDF 502
화면과 키보드에 대한 출력과 입력 및 UDF 502
확장 UNIX 코드(EUC)
고정 또는 가변 길이 데이터 유형 594
그래픽 데이터 처리 585
그래픽 상수 585
대만어 고려사항 584
대만어 코드 세트 581, 584
동일하지 않은 코드 페이지 587
문자 변환 오버플로우 594
문자 집합 578
문자열 길이 오버플로우 595
문자열 변환 규칙 595
서버에서의 확장 590
응용프로그램에서의 확장 590
일본어 및 대만어
COBOL 고려사항 785
FORTRAN 고려사항 803
일본어 코드 세트 581, 584
저장 프로시듀어 고려사항 586
저장 프로시듀어에서 문자 변환 596

확장 UNIX 코드(EUC) *(계속)*
조합시 고려사항 586
클라이언트에 기초한 매개변수 검증 591
혼합 코드 세트 환경 586
혼합 코드 페이지 583
확장 샘플 591
2바이트 코드 페이지 583
C/C++에서 대만어 700
C/C++에서 일본어 700
DBCLOB에 대한 고려사항 586
DESCRIBE문 사용 592
REXX에서 일본어 826
UDF 고려사항 585
확장된 동적 SQL문
DB2 Connect에서 지원 안함 899
확장성 및 구별 유형 303
환경 핸들 189
환경 API
COBOL에 대한 include 파일 763
C/C++용 include 파일 664
FORTRAN에 대한 include 파일 790
환경, 프로그래밍 12
활성화 시간 및 트리거 543

[숫자]

2바이트 문자 대형 오브젝트(LOB) 383
2바이트 문자 세트
구성 매개변수 579
대만어 고려사항 584
대만어 코드 세트 584
동일하지 않은 코드 페이지 587
일본어 코드 세트 584
조합시 고려사항 586
혼합 코드 세트 환경 586
2바이트 문자 세트(DBCS)
대만어 코드 세트 581
일본어 코드 세트 581

2바이트 문자 지원
특수 문자에 대한 코드 포인트 570
2바이트 코드 페이지 583
2진 대형 오브젝트(BLOB) 383
64비트 정수(BIGINT) 데이터 유형
DB2 Connect 버전 7에서 지원 885

A

ACQUIRE문 898
ActiveX 데이터 오브젝트 스펙
DB2에서 지원 31
ADD METHOD 322
ADHOC.SQC C 프로그램 목록 177
ADO 스펙
DB2에서 지원 31
AFTER 트리거 543, 549
ALLOW PARALLEL절 468
ALTER NICKNAME문
데이터 유형 맵핑 650
컬럼 옵션 646
API(Application Program Interface)
개요 44
스레드간 문맥 설정
sqleAttachToCtx() 608
sqleBeginCtx() 608
sqleDetachFromCtx() 608
sqleEndCtx() 608
sqleGetCurrentCtx() 608
sqleInterruptCtx() 608
sqleSetTypeCtx() 608
의 사용 44
의 유형 44
JDBC 응용프로그램 750
REXX에 대한 구문 821
XA 환경에서의 제한사항 618
APPC, 인터럽트 조절 136
ARI (VSE & VM용 DB2) 885
AS LOCATOR절 493
ASCII
정렬 순서 889
혼합 바이트 데이터 884
AS/400용 DB2 Universal Database
FOR BIT DATA 저장 프로시듀어 고려사항 247

B

BASIC 언어
OLE 자동화 UDF의 구현 472
BASIC 유형 및 OLE 자동 유형 475
BASIC으로 작성한 메일 OLE 자동 UDF 오브젝트 예제 532
BASIC으로 작성한 OLE 자동 UDF 오브젝트 예제 528
BEFORE 트리거 543, 549
BEGIN DECLARE SECTION 14
BigDecimal Java 유형 712
BIGINT SQL 데이터 유형 91
COBOL 781
C/C++ 702
FORTRAN 801
Java 712
Java 저장 프로시듀어 (DB2GENERAL) 863
OLE DB 테이블 함수 485
BIND API 69
BIND API, 패키지 작성 65
BIND PACKAGE 명령
리바인딩 72
blob C/C++ 유형 702
BLOB FORTRAN 유형 801
BLOB SQL 데이터 유형 91, 475
COBOL 781
C/C++ 702
FORTRAN 801
Java 712
Java 저장 프로시듀어 (DB2GENERAL) 863
BLOB SQL 데이터 유형 91, 475 *(계속)*
OLE DB 테이블 함수 485
REXX 817
BLOB(2진 대형 오브젝트)
사용 및 정의 383
BLOB에서의 문자열 검색 419
BLOB-FILE COBOL 유형 781
BLOB-LOCATOR COBOL 유형 781
blob_file C/C++ 유형 702
BLOB_FILE FORTRAN 유형 801
blob_locator C/C++ 유형 702
BLOB_LOCATOR FORTRAN 유형 801
BSTR OLE 자동 유형 475

C

C 널(NULL) 종료 그래픽 문자열 SQL 데이터 유형 475
C 널(NULL) 종료 문자열 SQL 데이터 유형 475
Call Level Interface (CLI)
개요 189
CALL USING DESCRIPTOR문(OS/400) 893
CALL문
상이한 플랫폼 893
저장 프로시듀어 호출 214
저장 프로시듀어(DB2DARI)를 위해 클라이언트 초기화
SQLDA 구조 858
Java 736
call-type 490
스칼라 함수를 포함하는 내용 444
테이블 함수를 포함하는 내용 445
call-type, UDF로 전달 443
CAST FROM절 436
CHAR 439
char C/C++ 유형 702
CHAR SQL 데이터 유형 91, 475

CHAR SQL 데이터 유형 91, 475 *(계속)*
  COBOL 781
  C/C++ 702
  FORTRAN 801
  Java 712
  Java 저장 프로시듀어
   (DB2GENERAL) 863
  OLE DB 테이블 함수 485
  REXX 817
CHAR 유형 750
CHARACTER*n FORTRAN 유형 801
CHECKERR REXX 프로그램 목록 143
CHECKERR.CBL 프로그램 목록 141
CICS 882
CICS SYNCPOINT ROLLBACK 명령 617
CLASSPATH 환경 변수 741
CLI 189
clob C/C++ 유형 702
CLOB FORTRAN 유형 801
CLOB SQL 데이터 유형 91, 475
  COBOL 781
  C/C++ 702
  FORTRAN 801
  Java 712
  Java 저장 프로시듀어
   (DB2GENERAL) 863
  OLE DB 테이블 함수xin2
  REXX 817
CLOB 값에 대해 작업할 위치 지정자 사용 예 388
CLOB 컬럼에 데이터 삽입 예 408
CLOB(문자 대형 오브젝트)
  사용 및 정의 383
CLOB-FILE COBOL 유형 781
CLOB-LOCATOR COBOL 유형 781
clob_file C/C++ 유형 702
CLOB_FILE FORTRAN 유형 801
clob_locator C/C++ 유형 702
CLOB_LOCATOR FORTRAN 유형 801
CLOSE 호출 445
COBOL
  오브젝트 지향 제한사항 786
  일본어 및 대만어 EUC 고려사항 785
  입력 및 출력 파일 761
  제한사항 761
  지원되는 데이터 유형 780
  파일 참조 선언 775
  표시기 변수에 대한 규칙 773
  표시기 테이블 778
  호스트 변수 선언 769
  embedding SQL문 55
  include 파일, 목록 762
  LOB LOB 위치 지정자 선언 774
  LOB 데이터 선언 773
COBOL 데이터 유형
  BINARY 780
  BLOB 781
  BLOB-FILE 781
  BLOB-LOCATOR 781
  CLOB 781
  CLOB-FILE 781
  CLOB-LOCATOR 781
  COMP 780
  COMP-1 781
  COMP-3 781
  COMP-4 780
  COMP-5 781
  DBCLOB 781
  DBCLOB-FILE 781
  DBCLOB-LOCATOR 781
  PICTURE(PIC)절 781
  USAGE절 781
COBOL 언어
  지원되는 데이터 유형 780
COBOL 유형의 COMP-1 781
COBOL 유형의 COMP-3 781
COBOL 유형의 COMP-5 781
COBOL 유형의 PICTURE(PIC)절 781
COBOL 유형의 USAGE절 781
COBOL에서 REDEFINES 779
COBOL의 BINARY 데이터 유형 780
COBOL의 COMP 및 COMP-4 데이터 유형 780
COLLECTION 매개변수 67
COMMIT WORK RELEASE문
  지원 안함 899
  DB2 Connect에서 지원 안함 899
COMMIT문 13
  커서와의 연결 97
  트랜잭션 종료 22, 23
  pass-through 657, 658
COM.ibm.db2.app.Blob 870
COM.ibm.db2.app.Clob 871
COM.ibm.db2.app.Lob 870
COM.ibm.db2.app.StoredProc 866
COM.ibm.db2.app.UDF 466, 867
COM.ibm.db2.jdbc.app.DB2Driver 718
COM.ibm.db2.jdbc.net.DB2Driver 718
CONNECT
  샘플 프로그램 122
  응용프로그램 20
  SQLCA.SQLERRD 설정 590
CONNECT RESET문
  트랜잭션 종료 23
CONNECT TYPE 2
  저장 프로시듀어에서의 고려사항 249
CONNECT문 13
CONNECT에서 SQLCA.SQLERRD 설정 590
CONVERT
  WCHARTYPE
   저장 프로시듀어의 248
CREATE DATABASE API
  SQLEDBDESC 구조 565

CREATE DISTINCT TYPE 사용 비용 예 305
CREATE DISTINCT TYPE 사용 재개 예 305
CREATE DISTINCT TYPE문
  구별 유형 정의 304
  및 유형변환 기능 412
  사용 예 305
CREATE FUNCTION MAPPING문
  연합 서버에 데이터 소스 함수 인식 653
  함수 이름 지정 656
  함수 호출의 오버헤드 감소 654
  함수를 호출하는 오버헤드 추정 655
CREATE FUNCTION문 442, 443, 446, 490, 493
  연합 시스템의 654
  CAST FROM절 453
  Java UDF 468
  OLE 자동 UDF 472
  RETURNS절 453
  UDF 등록 416
CREATE FUNCTION문의 CAST FROM절 453
CREATE FUNCTION문의 RETURNS절 453
CREATE METHOD 322
CREATE METHOD문
  메소드 등록 416
CREATE PROCEDURE문 214, 740
CREATE SERVER문 484
CREATE STORGROUP문
  DB2 Connect 지원 883
CREATE TABLESPACE문
  DB2 Connect 지원 883
CREATE TABLE문
  사용 예 306
  컬럼 옵션 정의 332
  LOB 컬럼 정의 385
  lob-options-clause 385
CREATE TABLE문 *(계속)*
  tablespace-options-clause 385
CREATE TABLE문의 lob-options-clause 385
CREATE TABLE문의 tablespace-options-clause 385
CREATE TABLE을 사용한 응용프로그램 양식 예 306
CREATE TABLE을 사용한 판매 예 306
CREATE TRIGGER문
  개요 539
  복수 트리거 551
  트리거 활성화 순서 543
  REFERENCING절 546
CREATE TRIGGER문에서 REFERENCING절 546
CREATE TYPE
  구조화 유형 318
CREATE TYPE MAPPING문 649
CREATE TYPE문
  MODE DB2SQL절 316
  REF USING절 319
CREATE USER MAPPING문 484
CREATE VIEW문
  유형화 뷰 작성 338
ctr() UDF C 프로그램 목록 515
CURRENT EXPLAIN MODE 레지스터 66
CURRENT FUNCTION PATH 레지스터 66
CURRENT QUERY OPTIMIZATION 레지스터 66
CURSOR.SQB COBOL 프로그램 목록 105
CURSOR.SQC C 프로그램 목록 101
Cursor.sqlj Java 프로그램 목록 103
C++
  유형 장식물 고려사항 662
C++ *(계속)*
  저장 프로시듀어를 위한 고려사항 248
  UDF에 대한 고려사항 504
C++ 유형 및 OLE 자동 유형 475
C++로 작성한 계수기 OLE 자동 UDF 오브젝트 예제 530
C/C++ 데이터 유형
  널로 종료되는 문자 양식 702
  blob 702
  blob_file 702
  blob_locator 702
  char 702
  clob 702
  clob_file 702
  clob_locator 702
  dbclob 702
  dbclob_file 702
  dbclob_locator 702
  double 702
  float 702
  long 702
  long int 702
  long long 702
  long long int 702
  short 702
  short int 702
  sqldbchar 702
  sqlint64 702
  VARCHAR 구조 양식 702
  wchar_t 702
C/C++ 언어
  구성원 연산자, 제한사항 693
  규정 연산자, 제한사항 693
  그래픽 호스트 변수 선언 676
  널로 종료되는 문자열 조절 689
  대만어 EUC 고려사항 700
  데이터 유형의 포인터, C/C++에서 선언 690
  문자 세트 662

C/C++ 언어 *(계속)*
일본어 EUC 고려사항 700
입력 및 출력 파일 663
지원되는 데이터 유형 701
클래스 데이터 구성원 조절 692
파일 참조 선언 683
프로그래밍 제한사항 661
호스트 변수 선언 671
호스트 변수 초기화 684
호스트 변수, 명명 670
embedding SQL문 55
include 파일, 필수 663
LOB LOB 위치 지정자 선언 682
LOB 데이터 선언 679
trigraph 절차 662
C/C++ 언어의 매크로 처리 661
C/C++에서 규정 및 구성원 연산자 693

D

DATE OLE 자동 유형 475
DATE SQL 데이터 유형 91, 475
COBOL 781
C/C++ 702
FORTRAN 801
Java 712
Java 저장 프로시듀어 (DB2GENERAL) 863
OLE DB 테이블 함수 485
REXX 817
DB2 BIND 명령
패키지 작성 65
DB2 Call Level Interface (DB2 CLI)
데이터로의 액세스 고려사항 30
DB2 Connect 881
분리 레벨 892
DB2 Connect 프로그래밍 고려사항 881
DB2 Connect를 사용하는 DB2 881
DB2 Connect를 사용하는 OS/400 881
DB2 Connect를 사용하는 SQL/DS 881
DB2 PREP 명령
개요 60
예 60
DB2 라이브러리
구조 907
마법사 923
문서 서버 설정 924
온라인 도움말 919
온라인 정보 검색 925
온라인 정보 보기 921
인쇄된 책 주문 917
정보 센터 922
책 907
책에 대한 언어 식별자 915
최신 정보 916
PDF 책 인쇄 917
DB2 시스템 제어기 249
DB2 응용프로그램 설계, 안내 26
DB2 응용프로그램의 연결 20
DB2 프로그래밍용 모델 25
DB2 프로그램
설정 14
DB2 프로그램 설정 14
DB2Appl.java
응용프로그램 예 722
DB2ARXCS.BND REXX 바인드 파일 820
DB2ARXNC.BND REXX 바인드 파일 820
DB2ARXRR.BND REXX 바인드 파일 820
DB2ARXRS.BND REXX 바인드 파일 820
DB2ARXUR.BND REXX 바인드 파일 820
db2bfd, 바인드 파일 덤프 유틸리티 69
DB2CODEPAGE 레지스트리 변수 566
db2dari 실행파일 233
DB2DARI 저장 프로시듀어 221
DB2DARI 저장 프로시듀어의 제한사항 859
db2dclgn 명령 86
db2diag.log 파일 637
DB2GENERAL 저장 프로시듀어 221
DB2INCLUDE 환경 변수 667, 767, 793
db2nodes.cfg 파일 639
db2udf 실행가능 파일 502
DB2Udf.java 466
DB2_SQLROUTINE_KEEP_FILES 282
DBCLOB
대만어 코드 세트 586
사용 및 정의 383
일본어 코드 세트 586
dbclob C/C++ 유형 702
DBCLOB SQL 데이터 유형 91, 475
COBOL 781
C/C++ 702
Java 712
Java 저장 프로시듀어 (DB2GENERAL) 863
OLE DB 테이블 함수 485
REXX 817
DBCLOB 유형 750
DBCLOB-FILE COBOL 유형 781
DBCLOB-LOCATOR COBOL 유형 781
dbclob_file C/C++ 유형 702
dbclob_locator C/C++ 유형 702
DBCS 581
dbinfo 490
DBINFO 키워드 446
dbinfo, UDF로 전달 446
dbminfo 인수, 요소 446
고유 응용프로그램 식별자 (appl_id) 450
데이터베이스 이름 길이 (dbnamelen) 447
데이터베이스 이름(dbname) 447

dbminfo 인수, 요소 446 *(계속)*
데이터베이스 코드 페이지 (codepg) 447
버전/릴리스 번호(ver_rel) 448
스키마 이름 길이(tbschemalen) 447
스키마 이름(tbschema) 447
응용프로그램 권한 부여 ID 길이 (authidlen) 447
응용프로그램 권한 부여 ID(authid) 447
컬럼 이름 길이(colnamelen) 448
컬럼 이름(colname) 448
테이블 이름 길이(tbnamelen) 448
테이블 이름(tbname) 448
테이블 함수 컬럼 목록 (tfcolumn) 449
테이블 함수 컬럼 항목 (numtfcol) 449
플랫폼 449
appl_id(고유 응용프로그램 식별자) 450
authidlen(응용프로그램 권한 부여 ID 길이) 447
authid(응용프로그램 권한 부여 ID) 447
codepg(데이터베이스 코드 페이지) 447
colnamelen(컬럼 이름 길이) 448
colname(컬럼 이름) 448
dbnamelen(데이터베이스 이름 길이) 447
dbname(데이터베이스 이름) 447
numtfcol(테이블 함수 컬럼 항목) 449
tbnamelen(테이블 이름 길이) 448
tbname(테이블 이름) 448
tbschemalen(스키마 이름 길이) 447
tbschema(스키마 이름) 447
tfcolumn(테이블 함수 컬럼 목록) 449

dbminfo 인수, 요소 446 *(계속)*
(ver_rel)버전/릴리스 번호 448
DCL(data control language) 885
DDL 883
DDL(Data Definition Language)
저장점에서 발행 202
DDL(data definition language) 883
DECIMAL SQL 데이터 유형 91, 475
COBOL 781
C/C++ 702
FORTRAN 801
Java 712
Java 저장 프로시듀어 (DB2GENERAL) 863
OLE DB 테이블 함수 485
REXX 817
DECLARE CURSOR문 20
DECLARE CURSOR문, 의 개요 96
DECLARE PROCEDURE문(OS/400) 893
DECLARE STATEMENT
DB2 Connect 지원 899
DECLARE문 898
DELETE
트리거 545
DB2 Connect 지원 883
DELETE 조작 및 트리거 540
DEREF 기능
정의 344
DEREF 함수
필수 특권 345
DESCRIBE문 898
구조화 유형 382
임의 명령문 처리 172
확장 UNIX 코드 고려사항 592
2바이트 문자 세트 고려사항 592
DB2 Connect 지원 899
EUC 데이터베이스 사용시 확장 UNIX 코드 고려사항 593

DFT_SQLMATHWARN 구성 매개변수 438
diagnostic-message, UDF로 전달 441
divid() UDF C 프로그램 목록 506
DML(data manipulation language) 883
double C/C++ 유형 702
double Java 유형 712
double OLE 자동 유형 475
DOUBLE SQL 데이터 유형 91, 475
DROP문
사용자 정의 유형 340
유형 맵핑 340
DSN (OS/390용 DB2 Universal Database) 885
DSS (분산 서브섹션) 622
DUOW 598
DYNAMIC.CMD REXX 프로그램 목록 160
Dynamic.java Java 프로그램 목록 156
DYNAMIC.SQB COBOL 프로그램 목록 158
DYNAMIC.SQC C 프로그램 목록 154

E

EBCDIC
정렬 순서 889
혼합 바이트 데이터 884
embedded SQL
데이터로의 액세스 고려사항 28
embedded SQL문
개요 55
구문 규칙 56
규칙, COBOL 765
규칙, C/C++ 668
규칙, FORTRAN 793
예 56
주석, 규칙 669, 766, 793
호스트 변수, 참조 88
END DECLARE SECTION 14

EUC 581
EUC(확장 UNIX 코드)
문자 집합 578
EXCSQLSTT 명령 898
EXEC SQL INCLUDE SQLCA
다중 스레드 고려사항 608
EXEC SQL INCLUDE문, C/C++ 언어 제한사항 667
EXECUTE IMMEDIATE문, 의 요약 146
EXECUTE문, 의 요약 146
exit 루틴, 사용시 제한사항 137
Explain 스냅샷 67
EXPLAIN, 유틸리티 프로토타입 51
EXPLSNAP 바인드 옵션 67
extern declaration
C++ 662
EXTERNAL ACTION 옵션 및 UDF 499
EXTERNAL NAME절 482, 484
EXTERNAL절 216

F

FENCED 옵션 및 UDF 499
FETCH 호출 445
FETCH문
뒤로 이동, 기술 118
반복된 액세스, 기술 118
호스트 변수 150
SQLDA 구조 사용 166
FINAL CALL 키워드 443
FINAL CALL절 445
findvwl() UDF C 프로그램 목록 510
FIPS 127-2 표준 18
FIRST 호출 445
float C/C++ 유형 702
float OLE 자동 유형 475
FLOAT SQL 데이터 유형 91, 475
COBOL 781
C/C++ 702
FLOAT SQL 데이터 유형 91, 475 *(계속)*
FORTRAN 801
Java 712
Java 저장 프로시듀어 (DB2GENERAL) 863
OLE DB 테이블 함수 485
REXX 817
fold() UDF C 프로그램 목록 510
FOR BIT DATA
저장 프로시듀어 고려사항 247
C/C++의 데이터 유형 708
FOR BIT DATA SQL 데이터 유형 475
FOR EACH ROW 트리거 542
FOR EACH STATEMENT 트리거 542
FOR UPDATE절 107
FORCE 명령 886
FORTRAN
일본어 및 대만어 EUC 고려사항 803
입력 및 출력 파일 788
지원되는 데이터 유형 800, 802
파일 참조 선언 800
표시기 변수, 규칙 798
프로그래밍 고려사항 787
프로그래밍 제한사항 788
호스트 변수 참조 794
호스트 변수, 개요 794
embedding SQL문 55
LOB LOB 위치 지정자 선언 799
LOB 데이터 선언 798
FORTRAN 데이터 유형
BLOB 801
BLOB_FILE 801
BLOB_LOCATOR 801
CHARACTER*n 801
CLOB 801
CLOB_FILE 801
FORTRAN 데이터 유형 *(계속)*
CLOB_LOCATOR 801
INTEGER*2 801
INTEGER*4 801
REAL*2 801
REAL*4 801
REAL*8 801
FORTRAN 언어
지원되는 데이터 유형 800
FROM SQL 변환 359
fullselect
버퍼 삽입 고려사항 630
fullselect 고려사항 630
FUNCPATH 바인드 옵션 67
function-name, UDF로 전달 440

G

GENERAL WITH NULLS 저장 프로시듀어 221
GENERAL 저장 프로시듀어 221
GET ERROR MESSAGE API 137, 811
getAsciiStream JDBC 메소드 750
getString JDBC 메소드 750
getUnicodeStream JDBC 메소드 750
GRANT문
테이블 계층에서 발행 331
DB2 Connect 지원 885
GRAPHIC SQL 데이터 유형
COBOL 781
C/C++ 702
FORTRAN, 지원 안함 801
Java 712
Java 저장 프로시듀어 (DB2GENERAL) 863
OLE DB 테이블 함수 485
REXX 817
GRAPHIC 유형 750
GROUP BY절
정렬 순서 889

H

HTML
- 샘플 프로그램 915

HTML 페이지
- Java 애플릿의 태그 지정 722

I

IN 저장 프로시듀어 매개변수 215, 229

INCLUDE SQLCA
- 의사 코드 17

INCLUDE SQLDA문 20

INCLUDE SQLDA문, SQLDA 구조 작성 167

include 파일
- COBOL 요긋사항 762
- COBOL로 찾기 767
- C/C++ 요구사항 663
- C/C++에서 찾기 667
- C/C++용 SQL 663
- C/C++용 SQLADEF 663
- C/C++용 SQLAPREP 663
- C/C++용 SQLCA 663
- C/C++용 SQLCLI 664
- C/C++용 SQLCLI1 664
- C/C++용 SQLCODES 664
- C/C++용 SQLDA 664
- C/C++용 SQLE819A 665
- C/C++용 SQLE819B 665
- C/C++용 SQLE850A 665
- C/C++용 SQLE850B 665
- C/C++용 SQLE932A 665
- C/C++용 SQLE932B 665
- C/C++용 SQLEAU 664
- C/C++용 SQLENV 664
- C/C++용 SQLEXT 664
- C/C++용 SQLJACB 666
- C/C++용 SQLMON 666
- C/C++용 SQLSTATE 666

include 파일 *(계속)*
- C/C++용 SQLSYSTM 666
- C/C++용 SQLUDF 666
- C/C++용 SQLUTIL 666
- C/C++용 SQLUV 666
- C/C++용 SQLUVEND 666
- C/C++용 SQLXA 666
- FORTRAN 요구사항 789
- FORTRAN에서 찾기 792
- SQL
  - COBOL 762
  - FORTRAN 789
- SQL1252A
  - COBOL 764
  - FORTRAN 791
- SQL1252B
  - COBOL 764
  - FORTRAN 791
- SQLAPREP
  - COBOL 762
  - FORTRAN 789
- SQLCA
  - COBOL 762
  - FORTRAN 789
- SQLCA_92
  - COBOL 762
  - FORTRAN 789
- SQLCA_CN
  - FORTRAN 789
- SQLCA_CS
  - FORTRAN 789
- SQLCODES
  - COBOL 763
  - FORTRAN 789
- SQLDA
  - COBOL 763
  - FORTRAN 790
- SQLDACT
  - FORTRAN 790

include 파일 *(계속)*
- SQLE819A
  - COBOL 763
  - FORTRAN 790
- SQLE819B
  - COBOL 763
  - FORTRAN 790
- SQLE850A
  - COBOL 764
  - FORTRAN 790
- SQLE850B
  - COBOL 764
  - FORTRAN 791
- SQLE932A
  - COBOL 764
  - FORTRAN 791
- SQLE932B
  - COBOL 764
  - FORTRAN 791
- SQLEAU
  - COBOL 763
  - FORTRAN 790
- SQLENV
  - COBOL 763
  - FORTRAN 790
- SQLETSD
  - COBOL 763
- SQLMON
  - COBOL 765
  - FORTRAN 791
- SQLMONCT
  - COBOL 765
- SQLSTATE
  - COBOL 765
  - FORTRAN 791
- SQLUTBCQ
  - COBOL 765
- SQLUTBSQ
  - COBOL 765

include 파일 *(계속)*
SQLUTIL
COBOL 765
FORTRAN 792
include 파일 찾기
COBOL 767
C/C++ 667
FORTRAN 792
INCLUDE문 20
infix 표기 및 UDF 429
INHERIT SELECT PRIVILEGES절 331
INOUT 저장 프로시듀어 매개변수 215, 230
INSERT BUF 바인드 옵션
버퍼 삽입 627
INSERT 조작 및 트리거 540
INSERT문
에 대해 유형화 테이블의 데이터 처리 중 333
CLP에서 지원되지 않음 630
DB2 Connect 지원 883, 884
VALUES절을 사용 626
Int Java 유형 712
INTEGER 443
INTEGER SQL 데이터 유형 91, 475
COBOL 781
C/C++ 702
FORTRAN 801
Java 712
Java 저장 프로시듀어 (DB2GENERAL) 863
OLE DB 테이블 함수 485
REXX 817
INTEGER*2 FORTRAN 유형 801
INTEGER*4 FORTRAN 유형 801
IS OF 술어
로 리턴된 유형의 제한 345
ISO 10646 표준 581
ISO 2022 표준 581
ISO/ANS SQL92 889
ISO/ANS SQL92 표준 18

J

Java
개요 709
다른 언어와 비교 710
디버깅 715
애플릿 분배 및 수행 722
애플릿 지원 717
연결 풀링 725
응용프로그램 분배 및 수행 722
응용프로그램 지원 716
저장 프로시듀어 746, 747
예 749
트랜잭션 API (JTA) 725
DB2 지원에 대한 개요 716
embedding SQL문 55
JAR 파일 설치 746, 747
JDBC 2.0 선택적 패키지 API 지원 724
JDBC 스펙 715
JDBC 예제 프로그램 719
JNDI 지원 724
SQLCODE 714
SQLJ (Embedded SQL for Java)
예제 프로그램 사용 732
저장 프로시듀어 호출 736
호스트 변수 736
SQLJ (Embedded SQLJ for Java) 726
변환 프로그램 727
애플릿 728
예제 절 730
제한사항 728
db2profc 727
db2profp 727
Embedded SQL문 730
profconv 727
SQLJ 스펙 715
Java *(계속)*
SQLJ(Embedded SQLJ for Java)
리턴 가능성 730
반복기 선언 730
보유 가능성 730
위치 지정된 DELETE문 730
위치 지정된 UPDATE문 730
커서 선언 730
SQLJ와 JDBC 비교 710
SQLMSG 714
SQLSTATE 714
UDF(사용자 정의 함수) 746, 747
예 749
Java data 유형
BigDecimal 712
Blob 712
double 712
Int 712
java.math.BigDecimal 712
short 712
String 712
Java I/O 스트림
System.err 466
System.in 466
System.out 466
Java UDF 고려사항 435
Java UDF 코딩 466
Java 메소드 마무리 468
Java 명명 및 디렉토리 인터페이스 (JNDI) 724
Java 응용프로그램
그래픽 및 대형 오브젝트(LOB) 사용 750
SCRATCHPAD 고려사항 468
UDF에 대한 시그니쳐 466
JAVA 저장 프로시듀어 221
Java 저장 프로시듀어 등록 740
Java 클래스 파일
위치 741
CLASSPATH 환경 변수 741

Java 클래스 파일 *(계속)*
java_heap_sz 구성 매개변수 741
jdk11_path 구성 매개변수 741
Java 패키지 및 클래스 718
COM.ibm.db2.app 712
Java에서 연결 풀링 725
java.math.BigDecimal Java 유형 712
java_heap_sz 구성 매개변수 741
JDBC
데이터로의 액세스 고려사항 30
예제 프로그램 사용 719
1.22 드라이버 723
2.0 드라이버 723
2.0 선택적 패키지 API 지원 724
2.0 코어 API 723
COM.ibm.db2.jdbc.app.
DB2Driver 718
COM.ibm.db2.jdbc.net.
DB2Driver 718
getAsciiStream 메소드 750
getString 메소드 750
getUnicodeStream 메소드 750
setAsciiStream 메소드 750
setString 메소드 750
setUnicodeStream 메소드 750
SQLJ 상호운용성 752
SQLJ와 비교 710
JDBC(Java Database
Connectivity) 718
jdk11_path 구성 매개변수 741
JNDI(Java 명명 및 디렉토리 인터페이
스) 724
JTA (Java Transaction API) 725

L

LABEL ON문 898
LANGLEVEL SQL92E 사전 처리 컴파
일 옵션 889
LANGLEVEL 사전 처리 컴파일 옵션
MIA 707
LANGLEVEL 사전 처리 컴파일 옵션
(계속)
SAA1 707
SQL92E 및 SQLSTATE 또는
SQLCODE 변수 사용 708, 785,
803
LANGUAGE OLE절 472
LOB 데이터 유형
DB2 Connect 버전 7에서 지원 884
LOB 위치 지정자
사용 시나리오 498
UDF에서 사용 493
LOB 위치 지정자 예제 프로그램 목록
525
LOB 위치 지정자 API, UDF에서 사용
sqludf_append API 493
sqludf_create_locator API 493
sqludf_free_locator API 493
sqludf_length API 493
sqludf_substr API 493
LOB 조작용 위치 지정자 383
LOB 표현식 평가 연기 예 394
LOBEVAL.SQB COBOL 프로그램 목록
398, 406
LOBEVAL.SQC C 프로그램 목록 396,
404
LOBLOC.SQB COBOL 프로그램 목록
391
LOBLOC.SQC C 프로그램 목록 389
LOCKTIMEOUT 구성 매개변수 612
LOCKTIMEOUT 다중 사이트 갱신 구성
매개변수 604
long C/C++ 유형 702
long int C/C++ 유형 702
long long C/C++ 유형 702
long long int C/C++ 유형 702
long OLE 자동 유형 475
LONG VARCHAR
저장영역 한계 383
UDF로의 매개변수 458
LONG VARCHAR SQL 데이터 유형
91, 475
COBOL 781
C/C++ 702
FORTRAN 801
Java 712
Java 저장 프로시듀어
(DB2GENERAL) 863
OLE DB 테이블 함수 485
REXX 817
LONG VARGRAPHIC
저장영역 한계 383
UDF로의 매개변수 460
LONG VARGRAPHIC SQL 데이터 유
형 91, 475
COBOL 781
C/C++ 702
FORTRAN 801
Java 712
Java 저장 프로시듀어
(DB2GENERAL) 863
OLE DB 테이블 함수 485
REXX 817
long 필드 제한사항
버퍼 삽입 사용 630
LONGVAR 유형 750

M

maxdari 구성 매개변수 740
MIA 707
Microsoft Exchange, 메일 예제에 사용
532
Microsoft Visual C++
IBM DB2 Universal Database 프로
젝트 추가 37
Microsoft Visual C++에 대한 IBM
DB2 Universal Database 도구 추가,
활성화 40

Microsoft Visual C++에 대한 IBM DB2 Universal Database 프로젝트 추가 37, 40
Microsoft 스펙
 데이터로의 액세스 고려사항 31
 ADO(ActiveX Data Object) 31
 MTS(Microsoft Transaction Server) 31
 RDO(Remote Data Object) 31
 Visual Basic 31
 Visual C++ 31
Microsoft 트랜잭션 서버 스펙
 데이터로의 액세스 고려사항 31
MODE DB2SQL절 316

N

Netscape 브라우저
 설치 921
NOCONVERT
 WCHARTYPE
  저장 프로시듀어의 248
NOLINEMACRO
 PREP 옵션 668
NOT DETERMINISTIC 옵션 및 UDF 499
NOT FENCED LOB 위치 지정자 UDF 493
NOT FENCED 저장 프로시듀어
 고려사항 250
 사전 처리 컴파일 250
 작업 249
NOT NULL CALL 옵션 및 UDF 499
NOT NULL CALL절 437
NUMERIC SQL 데이터 유형 475
 COBOL 781
 C/C++ 702
 FORTRAN 801
 Java 712
 Java 저장 프로시듀어 (DB2GENERAL) 863
NUMERIC SQL 데이터 유형 475 *(계속)*
 OLE DB 테이블 함수 485
 REXX 817

O

ODBC
 데이터로의 액세스 고려사항 30
OLE DB
 서버 이름 사용 481
 연결 문자열 사용 482
 테이블 함수 479
  사용자 맵핑 정의 484
  서버 식별 484
  완전한 이름 482
  작성 480
  CREATE SERVER문 484
  CREATE USER MAPPING문 484
  EXTERNAL NAME절 482
 CONNECTSTRING 옵션 사용 481
 DB2에서 지원 31
OLE DB 테이블 함수의 USER MAPPING 484
OLE 메시지의 예 532
OLE 자동 데이터 유형 475
 BSTR 475
 DATE 475
 double 475
 float 475
 long 475
 SAFEARRAY 475
 short 475
OLE 자동 서버 472
OLE 자동 유형 474
OLE 자동 유형 및 BASIC 유형 475
OLE 자동 유형 및 C++ 유형 475
OLE 자동 UDF
 구현 472
OLE 자동 UDF *(계속)*
 스크래치 패드 고려사항 473
 오브젝트 인스턴스 473
 작성가능한 단일 사용 OLE 자동 서버 478
 작성가능한 복수 사용 OLE 자동 서버 478
 BASIC으로 구현 476
 C++로 구현 476
 UDF 471
OLE 자동화 오브젝트 카운팅 예 422
OLE 자동화 오브젝트에서의 카운팅 422
OLE 자동화 UDF에 대한 프로그램 ID(progID) 472
OLE 키워드 471
OLE 프로그램 ID(progID) 472
ONLY절
 리턴된 유형 제한 345
OPENFTCH.SQB COBOL 프로그램 목록 116
OPENFTCH.SQC C 프로그램 목록 111
Openftch.sqlj Java 프로그램 목록 113
ORDER BY절
 정렬 순서 889
OS/390용 DB2 Universal Database
 FOR BIT DATA 저장 프로시듀어
  고려사항 247
OUT 저장 프로시듀어 매개변수 215, 229
OUTER 키워드
 부속 유형 속성 리턴 346

P

pass-through
 고려사항, 제한사항 657
 COMMIT문 657, 658
 SET PASSTHRU RESET문 658

pass-through *(계속)*
SET PASSTHRU문 657
SQL 처리 656
PDF 917
PDF 책 인쇄 917
Perl
데이터로의 액세스 고려사항 31
PREP 옵션
NOLINEMACRO 668
PREPARE문
의 요약 146
임의 명령문 처리 172
DB2 Connect 지원 899
PUT문
DB2 Connect에서 지원 안함 899

Q

QSQ (AS/400용 DB2 Universal Database) 885
QUERYOPT 바인드 옵션 67

R

RAISE_ERROR 내장 함수 549
RDO 스펙
DB2에서 지원 31
REAL SQL 데이터 유형 91, 475
COBOL 781
C/C++ 702
FORTRAN 801
Java 712
Java 저장 프로시듀어 (DB2GENERAL) 863
OLE DB 테이블 함수 485
REXX 817
REAL*2 FORTRAN 유형 801
REAL*4 FORTRAN 유형 801
REAL*8 FORTRAN 유형 801
REF USING절 319
RELEASE SAVEPOINT문 201
REORGANIZE TABLE 명령 583
RESULT REXX 사전 정의된 변수 811
RESYNC_INTERVAL 다중 사이트 갱신 구성 매개변수 604
RETURNS TABLE절 436
REVOKE문
명령문 885
테이블 계층에서 발행 331
DB2 Connect 지원 885
REXX
대만어 고려사항 826
데이터로의 액세스 고려사항 29
루틴 등록 806
바인드 파일 820
변수 초기화 823
사전 정의된 변수 811
실행 요구사항 819
응용프로그램에서 DB2 CLP 호출 821
일본어 고려사항 826
저장 프로시듀어 823
제한사항 806
지원되는 데이터 유형 817
지원되는 SQL문 808
커서 식별자 809
표시기 변수 811, 819
프로그래밍 고려사항 806
API 구문 821
LOB LOB 위치 지정자 선언 814
LOB 데이터 813
LOB 파일 참조 선언 815
LOB 호스트 변수 지움 816
SQLEXEC, SQLDBS 및 SQLDB2 등록 806
REXX API
SQLDB2 805, 821
SQLDBS 805
SQLEXEC 805
REXX 및 C++ 데이터 유형 817
REXX 응용프로그램에서 호출 821
REXX 응용프로그램에서 DB2 CLP 호출 821
REXX에 대한 바인드 파일 820
REXX에 대한 실행 요구사항 819
REXX에서 커서 사용 819
REXX에서 파일 참조 선언 815
REXX에서 SQL문 처리 808
re-entrant
저장 프로시듀어 250
ROLLBACK TO SAVEPOINT문 201
ROLLBACK WORK RELEASE
DB2 Connect에서 지원 안함 899
ROLLBACK문 13, 886
데이터 복원 23
변경사항 구간 복원 23
변경사항 보관 23
커서와의 연결 97
트랜잭션 종료 23
ROWID 데이터 유형
DB2 Connect 버전 7에서 지원 885
RQRIOBLK 필드 887
RUOW 597

S

SAA1 707
SAFEARRAY OLE 자동 유형 475
SAVEPOINT문 200
SCRATCHPAD 옵션
OLE 자동 UDF 473
SCRATCHPAD 키워드 442, 443, 468, 488
SCRATCHPAD절 445
scratchpad, UDF로 전달 442
SELECT INTO문
개요 78
SELECT문
검색된 데이터 갱신 121
다중 행 검색 95
데이터 두 번째 검색 119
버퍼 삽입 고려사항 628

SELECTL문 *(계속)*
설명, SQLDA 할당 이후 166
수퍼 테이블에서 특권 계승 331
유형화 테이블 342
의 범위 지정된 참조 342
의 참조 해제 연산자 342
DB2 Connect 지원 883
DECLARE CURSOR문의 96
EXECUTE문과의 연결 146
SQLDA 선언 162
varying-list, 개요 173
SET CURRENT FUNCTION PATH문 415
SET CURRENT PACKAGESET문 67
SET CURRENT문
DB2 Connect 지원 899
SET PASSTHRU RESET문 658
SET PASSTHRU문 657, 658
SET SERVER OPTION문 653
setAsciiStream JDBC 메소드 750
setString JDBC 메소드 750
setUnicodeStream JDBC 메소드 750
shift-out 및 shift-in 문자 884
short C/C++ 유형 702
short int C/C++ 유형 702
short Java 유형 712
short OLE 자동 유형 475
SIGUSR1 인터럽트 136
SIMPLE WITH NULLS 저장 프로시듀어 221
SIMPLE 저장 프로시듀어 221
SMALLINT 437, 438
SMALLINT SQL 데이터 유형 91, 475
COBOL 781
C/C++ 702
FORTRAN 801
Java 712
Java 저장 프로시듀어 (DB2GENERAL) 863
SMALLINT SQL 데이터 유형 91, 475 *(계속)*
OLE DB 테이블 함수 485
REXX 817
SmartGuides
마법사 923
specific-name, UDF로 전달 440
SPM_LOG_SIZE 다중 사이트 갱신 구성 매개변수 605
SPM_NAME 다중 사이트 갱신 구성 매개변수 604
SPM_RESYNC_AGENT_LIMIT 다중 사이트 갱신 구성 매개변수 604
SQL
권한 부여 고려사항 41
동적 SQL에 대한 권한 부여 고려사항 42
동적으로 준비 190
명령문, 요약 41
정적 SQL에 대한 권한 부여 고려사항 43
지원되는 명령문 테이블 829
API를 사용한 권한 부여 고려사항 43
SQL include 파일
COBOL 응용프로그램 762
C/C++ 응용프로그램 663
FORTRAN 응용프로그램 789
SQL 데이터 유형 474, 475
BIGINT 91
BLOB 91, 475
CHAR 91, 475
CLOB 91, 475
COBOL 781
C/C++로의 변환 702
DATE 91, 475
DBCLOB 91, 475
DECIMAL 91
DOUBLE 475
FLOAT 91, 475
SQL 데이터 유형 474, 475 *(계속)*
FOR BIT DATA 475
FORTRAN 801
GRAPHIC 475
INTEGER 91, 475
Java 712
LONG GRAPHIC 475
LONG VARCHAR 91, 475
LONG VARGRAPHIC 91, 475
NUMERIC 475
OLE DB 테이블 함수 485
REAL 91, 475
REXX 817
SMALLINT 91, 475
TIME 91, 475
TIMESTAMP 91, 475
VARCHAR 91, 475
VARGRAPHIC 91, 475
SQL 데이터 유형, UDF로 전달 453
SQL 선언 절 14
SQL 인수, DB2에서 UDF로 전달 435
SQL 코드 프로토타입 50
SQL 통신 영역(SQLCA) 17
SQL 프로시듀어
결과 세트 리턴 274, 276
결과 세트 수신 277
동적 SQL 272
디버깅 279, 282
로그 파일 282
반복 275
제한사항 275
조건 핸들러 268
CALL문 274
RESIGNAL 271
SIGNAL 271
SQL1252A include 파일
COBOL 응용프로그램 764
FORTRAN 응용프로그램 791
SQL1252B include 파일
COBOL 응용프로그램 764

SQL1252B include 파일 *(계속)*
FORTRAN 응용프로그램 791
SQL92 889
SQLADEF include 파일
C/C++ 응용프로그램 663
SQLAPREP include 파일
COBOL 응용프로그램 762
C/C++ 응용프로그램 663
FORTRAN 응용프로그램 789
SQLCA
다중 스레드 고려사항 608
다중 정의 피하기 19
버퍼 삽입에서의 오류 보고 628
오류 발생시 불완전한 삽입 628
SQLERRMC 필드 886, 897
SQLERRP 필드 885
SQLCA include 파일
COBOL 응용프로그램 762
C/C++ 응용프로그램 663
FORTRAN 응용프로그램 789
SQLCA 구조
개요 133
경고 90
다중 정의 135
병합된 복수 구조 637
오류 보고 638
요구사항 133
저장 프로시듀어 사용 861
정의, 샘플 프로그램 122
C/C++용 include 파일 663
include 파일
COBOL 응용프로그램 762
FORTRAN 응용프로그램 789
sqlerrd 637
SQLERRD(6) 필드 639
SQLWARN1 필드 90
SQLCA 사전 정의된 변수 811
SQLCA의 다중 정의, 피하기 19
SQLCA의 SQLERRMC 필드 576, 886, 897
SQLCA의 SQLERRP 필드 885
SQLCA_92 include 파일
COBOL 응용프로그램 762
FORTRAN 응용프로그램 789
SQLCA_92 구조
include 파일
FORTRAN 응용프로그램 789
SQLCA_CN include 파일 789
SQLCA_CS include 파일 789
SQLCHAR 구조
데이터 전달 171
SQLCLI include 파일
C/C++ 응용프로그램 664
SQLCLI1 include 파일
C/C++ 응용프로그램 664
SQLCODE
구조 133
독립형 889
오류 보고 638
플랫폼 차이점 891
Java 프로그램 714
SQLCA 포함 18
SQLCODE -1015 637
SQLCODE -1034 637
SQLCODE -1224 637
SQLCODES include 파일
COBOL 응용프로그램 763
C/C++ 응용프로그램 664
FORTRAN 응용프로그램 789
SQLCODE와 SQLSTATE 사이의 차이점 891
SQLDA
다중 스레드 고려사항 608
SQLDA include 파일
COBOL 응용프로그램 763
C/C++ 응용프로그램 664
FORTRAN 응용프로그램 790
SQLDA 구조
데이터 블록 전달에 사용 620
데이터 전달 171
SQLDA 구조 *(계속)*
사용, 샘플 프로그램 174
서버 입력 프로시듀어, 샘플 878
선언 162
입력-SQLDA 프로시듀어, 샘플 873
작성(할당) 167
작성, 호스트 언어 예 168
저장 프로시듀어 사용 861
저장 프로시듀어에 사용된 필드
SQLDATA 861
SQLIND 861
SQLLEN 861
SQLTYPE 861
저장 프로시듀어(DB2DARI)를 위해 초기화 858
최소를 사용하는 명령문 준비 163
충분한 SQLVAR 엔터티 선언 165
DB2DARI 저장 프로시듀어로 조작 859
PREPARE문과의 연결 146
SQLDACT include 파일 790
SQLDATA 필드 861
sqlda.n.SQLDAT 860
sqlda.n.SQLDATALEN 860
sqlda.n.SQLDATATYPE_NAME 860
sqlda.n.SQLIND 860
sqlda.n.SQLLEN 860
sqlda.n.SQLLONGLEN 860
sqlda.n.SQLNAME.data 860
sqlda.n.SQLNAME.length 860
sqlda.n.SQLTYPE 860
sqlda.SQLDABC 860
sqlda.SQLDAID 860
sqlda.SQLN 860
SQLDB2 REXX API 805, 821
SQLDB2, REXX에 대한 등록 806
sqldbchar C/C++ 유형 702
sqldbchar 데이터 유형 459, 460, 462, 695

sqldbchar 및 wchar_t, 데이터 유형 선택 695
SQLDBS REXX API 805
SQLDBS, REXX에 대한 등록 806
SQLE819A include 파일
  COBOL 응용프로그램 763
  C/C++ 응용프로그램 665
  FORTRAN 응용프로그램 790
SQLE819B include 파일
  COBOL 응용프로그램 763
  C/C++ 응용프로그램 665
  FORTRAN 응용프로그램 790
SQLE850A include 파일
  COBOL 응용프로그램 764
  FORTRAN 응용프로그램 790
SQLE850B include 파일
  COBOL 응용프로그램 764
  FORTRAN 응용프로그램 791
SQLE859A include 파일
  C/C++ 응용프로그램 665
SQLE859B include 파일
  C/C++ 응용프로그램 665
SQLE932A include 파일
  COBOL 응용프로그램 764
  C/C++ 응용프로그램 665
  FORTRAN 응용프로그램 791
SQLE932B include 파일
  COBOL 응용프로그램 764
  C/C++ 응용프로그램 665
  FORTRAN 응용프로그램 791
sqleAttachToCtx() API 608
SQLEAU include 파일
  COBOL 응용프로그램 763
  C/C++ 응용프로그램 664
  FORTRAN 응용프로그램 790
sqleBeginCtx() API 608
sqleDetachFromCtx() API 608
sqleEndCtx() API 608
sqleGetCurrentCtx() API 608
sqleInterruptCtx() API 608
SQLENV include 파일
  COBOL 응용프로그램 763
  C/C++ 응용프로그램 664
  FORTRAN 응용프로그램 790
SQLERRD(1) 576, 587, 589
SQLERRD(2) 576, 587, 589
SQLERRD(3)
  XA 환경 617
sqleSetTypeCtx() API 608
SQLETSD include 파일
  COBOL 응용프로그램 763
SQLException
  조절 140
  SQLCODE 양식 검색 714
  SQLMSG 검색 714
  SQLSTATE 양식 검색 714
SQLEXEC
  REXX에서 SQL문 처리 808
SQLEXEC REXX API 805
SQLEXEC, REXX에 대한 등록 806
SQLEXT include 파일
  CLI 응용프로그램 664
SQLIND 필드 861
sqlint64 C/C++ 유형 702
SQLISL 사전 정의된 변수 811
SQLJ (Embedded SQL for Java)
  개요 726
  애플릿 728
  예제 절 730
  예제 프로그램 사용 732
  저장 프로시듀어 호출 736
  제한사항 728
  호스트 변수 736
  db2profc 명령 727
  db2profp 명령 727
  Embedded SQL문 730
  JDBC(Java Database Connectivity) 상호운용성 752
  profconv 명령 727
  translator 명령 727
SQLJ (Embedded SQLJ for Java)
  JDBC(Java Database Connectivity) 와 비교 710
SQLJ 반복기에서의 리턴 가능성 730
SQLJ 반복기에서의 보유 가능성 730
SQLJACB include 파일
  C/C++ 응용프로그램 666
SQLJ(Embedded SQL for Java)
  리턴 가능성 730
  반복기 선언 730
  보유 가능성 730
  위치 지정된 DELETE문 730
  위치 지정된 UPDATE문 730
  커서 선언 730
SQLLEN 필드 861
SQLMON include 파일
  COBOL 응용프로그램 765
  C/C++ 응용프로그램 666
  FORTRAN 응용프로그램 791
SQLMONCT include 파일
  COBOL 응용프로그램 765
SQLMSG
  Java 프로그램 714
SQLMSG 사전 정의된 변수 811
SQLRDAT 사전 정의된 변수 811
SQLRIDA 사전 정의된 변수 811
SQLRODA 사전 정의된 변수 811
SQLSTATE
  독립형 889
  차이점 891
  CLI에서 190
  Java 프로그램 714
  SQLCA의 SQLERRMC 필드 897
SQLSTATE include 파일
  COBOL 응용프로그램 765
  C/C++ 응용프로그램 666
  FORTRAN 응용프로그램 791
SQLSTATE 필드, 오류 메시지 134
SQLSYSTM include 파일
  C/C++ 응용프로그램 666

SQLTYPE 필드 861
SQLUDF Include 파일
UDF 인터페이스 435
SQLUDF include 파일
설명 465
C/C++ 응용프로그램 666
sqludf.h include 파일 453
sqludf.h에서 구조 정의 465
sqludf.h에서 마크로 465
sqludf.h에서 C 언어 유형 정의 465
sqludf_append API 493
sqludf_create_locator API 493
sqludf_free_locator API 493
sqludf_length API 493
sqludf_substr API 493
SQLUTBCQ include 파일
COBOL 응용프로그램 765
SQLUTBSQ include 파일
COBOL 응용프로그램 765
SQLUTIL include 파일
COBOL 응용프로그램 765
C/C++ 응용프로그램 666
FORTRAN 응용프로그램 792
SQLUV include 파일
C/C++ 응용프로그램 666
SQLUVEND include 파일
C/C++ 응용프로그램 666
SQLVAR 엔티티
변수 번호, 선언 162
충분한 수 선언 165
SQLWARN 구조, 의 개요 134
SQLXA include 파일
C/C++ 응용프로그램 666
SQLZ_HOLD_PROC 리턴 값 862
SQLZ_연결해제 _PROC 리턴 값 862
SQL문
범주 881
신호 핸들러 136
예외 핸들러 136
인터럽트 핸들러 136
SQL문 *(계속)*
일반 사용자 요청 저장 174
저장 프로시듀어를 사용한 그룹화 210
COBOL 구문 765
C/C++ 구문 668
DB2 Connect 지원 898, 899
FORTRAN 구문 793
REXX 구문 808
REXX에서 지원됨 808
SQL문 실행
직렬화 607
SQL문 실행의 직렬화 607
SQL문으로 인터럽트 조절 136
SQL-argument 444
SQL-argument, UDF로 전달 436
SQL-argument-ind 444
SQL-argument-ind, UDF로 전달 437
SQL-result 444, 490
SQL-result, UDF로 전달 436
SQL-result-ind 444, 490
SQL-result-ind, UDF로 전달 438
SQL-state, UDF로 전달 439
SQL_API_FN 매크로 453, 859
SQL_FILE_READ, 입력 값 옵션 402
SQL_WCHART_CONVERT 선행 처리기 매크로 697
STATIC.SQB COBOL 프로그램 목록 82
STATIC.SQC C 프로그램 목록 80
Static.sqlj Java 프로그램 목록 81
Stored Procedure Builder
개요 895
기능 895
디버그 테이블 744
환경 설정 743
String 712
SYSCAT.FUNCMAPOPTIONS 카탈로그 뷰 654
SYSCAT.FUNCTIONS 카탈로그 뷰 654
SYSIBM.SYSPROCEDURES 카탈로그 (OS/390) 893
SYSSTAT.FUNCTIONS 카탈로그 뷰 655
System.err Java I/O 스트림 466
System.in Java I/O 스트림 466
System.out Java I/O 스트림 466

T

tfweather_u 테이블 함수 C 프로그램 목록 517
TIME SQL 데이터 유형 91, 475
COBOL 781
C/C++ 702
FORTRAN 801
Java 712
Java 저장 프로시듀어 (DB2GENERAL) 863
OLE DB 테이블 함수 485
REXX 817
TIMESTAMP SQL 데이터 유형 91, 475
COBOL 781
C/C++ 702
FORTRAN 801
Java 712
Java 저장 프로시듀어 (DB2GENERAL) 863
OLE DB 테이블 함수 485
REXX 817
TM_DATABASE 다중 사이트 갱신 구성 매개변수 604
TO SQL 변환 359
TP_MON_NAME 다중 사이트 갱신 구성 매개변수 604
TREAT 표현식 354
trigraph 절차 662

TYPE 술어
리턴된 유형 제한 345
TYPE_ID 함수 344
TYPE_NAME 함수 344
TYPE_SCHEMA 함수 344

U

UCS-2 581
UDF 구현 416
UDF 디버깅 535
UDF 링크 416
UDF 및 DB2간 인수 436
call-type 443
dbinfo 446
diagnostic-message 441
function-name 440
scratchpad 442
specific-name 440
SQL-argument 436
SQL-argument-ind 437
SQL-result 436
SQL-result-ind 438
SQL-state 439
UDF 및 LOB 유형 430
UDF 보호 499
UDF 예에서 정수 나누기 연산자 506
UDF 예에서의 계수기 515
UDF 컴파일 416
UDF 테스트 535
UDF 호출 424
UDF로 부동 소수점 매개변수 456
UDF로의 BIGINT 매개변수 455
UDF로의 BLOB 매개변수 462
UDF로의 CHAR 매개변수 457
UDF로의 CHARACTER 매개변수 457
UDF로의 CLOB 매개변수 462
UDF로의 DATE 매개변수 461
UDF로의 DBCLOB 매개변수 462
UDF로의 DECIMAL 매개변수 455
UDF로의 DOUBLE PRECISION 매개변수 457
UDF로의 DOUBLE 매개변수 455, 457
UDF로의 FLOAT 매개변수 457
UDF로의 GRAPHIC 매개변수 459
UDF로의 INTEGER 또는 INT 매개변수 455
UDF로의 NUMERIC 매개변수 455
UDF로의 REAL 매개변수 456
UDF로의 SMALLINT 매개변수 454
UDF로의 TIME 매개변수 461
UDF로의 TIMESTAMP 매개변수 461
UDF로의 VARCHAR FOR BIT DATA 매개변수 458
UDF로의 VARGRAPHIC 매개변수 460
UDF를 디버깅하기 위한 printf() 535
UDF(사용자 정의 함수)
개념 413
공유 메모리 크기 502
공유 메모리 크기에 대한 시스템 구성 매개변수 502
구현 410
구현 프로세스 416
규정화되지 않은 참조 413
대만어 코드 세트 585
등록 416
및 DB2 오브젝트 확장 297
보호 자원 사용시 고려사항 502
스크래치 패드 고려사항 488
스키마 이름 및 UDF 413
예 510
오버로드된 함수 이름 413
유형 목록 및 UDF에서의 표시 453
유형변환 430
이론적 근거 410
인수를 DB2에서 UDF로 전달 435
일반 고려사항 429
일본어 코드 세트 585
작성 416, 433
UDF(사용자 정의 함수) *(계속)*
재입력 UDF 488
전래 310
정의 409
제한사항 및 조건 502
조건 502
코드 페이지 차이점 502
코딩시 도움말 및 추가 정보 499
테이블 함수 490
트리거, UDT 및 LOB와의 시너지 553
함수 경로 413
함수 선택 알고리즘 413
함수 유형 414
함수 참조 424
함수 참조 요약 428
함수의 저장 상태 488
호출 424
규정화되지 않은 함수 참조 427
규정화된 함수 참조 426
함수의 매개변수 표시문자 426
호출 규약 453
화면과 키보드에 대한 입력과 출력 502
화면과 키보드에 대한 출력과 입력 502
C++ 고려사항 504
DB2 및 UDF간의 인터페이스 435
db2udf 실행가능 파일 502
DETERMINISTIC 489
EXTERNAL ACTION 옵션 499
FENCED 옵션 499
infix 표기 429
Java 고려사항 435
Java 작성 및 사용 466
Java 코딩 466
LOB 위치 지정자 사용 493
LOB 위치 지정자 사용법 시나리오 498
LOB 유형 430

UDF(사용자 정의 함수) *(계속)*
NOT DETERMINISTIC 488
NOT DETERMINISTIC 옵션 499
NOT FENCED 507
NOT NULL CALL 507
NOT NULL CALL 옵션 499
OLE 자동 UDF 471
SCRATCHPAD 489
SQL 데이터 유형, 전달되는 방식 453
SQLUDF Include 파일 435
SQLUDF include 파일 465
SQL_API_FN 507
SUBSTR 내장 함수 513
UDF 디버깅 535
UDF 코드 예 505
UDF에서 동적 메모리 할당 499
UDF에 대한 sqludf.h include 파일 465
UDF에서 동적 메모리 할당 499
UDF의 공유 메모리 크기 502
UDF의 FOR BIT DATA 수정자 457
UDT 비용 예 420
UDT(사용자 정의 유형)
및 DB2 오브젝트 확장 297
트리거, UDF 및 LOB와의 시너지 553
UDT에서의 문자열 검색 예 419
UDT의 AVG 예 421
UNION에서의 구별 유형 사용 예 314
UPDATE 조작 및 트리거 540
UPDATE문
DB2 Connect 지원 883
UPDATE, INSERT 또는 DELETE와 같은 트리거 이벤트 540
UPDAT.CMD REXX 프로그램 목록 131
UPDAT.SQB COBOL 프로그램 목록 129
UPDAT.SQC C 프로그램 목록 125
Updat.sqlj Java 프로그램 목록 127
UTILAPI.C 프로그램 목록 137

V

V5SPCLI.SQC C 프로그램 목록 876
V5SPSRV.SQC C 프로그램 목록 879
VALIDATE RUN
DB2 Connect 지원 886
VALUES절
INSERT문에서 626
VARCHAR 440, 441
VARCHAR SQL 데이터 유형 91, 475
C 또는 C++ 708
COBOL 781
C/C++ 702
FORTRAN 801
Java 712
Java 저장 프로시듀어 (DB2GENERAL) 863
OLE DB 테이블 함수 485
REXX 817
VARCHAR 구조 양식 C/C++ 유형 702
VARGRAPHIC SQL 데이터 유형 91, 475
COBOL 781
C/C++ 702
FORTRAN 801
Java 712
Java 저장 프로시듀어 (DB2GENERAL) 863
OLE DB 테이블 함수 485
REXX 817
VARGRAPHIC 데이터 707
Varinp.java Java 프로그램 목록 185
VARINP.SQB COBOL 프로그램 목록 187
VARINP.SQC C 프로그램 목록 183
Visual Basic
DB2에서 지원 31
Visual C++
DB2에서 지원 31
IBM DB2 Universal Database 프로젝트 추가 37

W

WCHARTYPE
저장 프로시듀어의 248
지침 697
WCHARTYPE 사전 처리 컴파일러 옵션 250, 696
wchar_t C/C++ 유형 702
wchar_t 데이터 유형 459, 460, 462, 695
wchar_t 및 sqldbchar, 데이터 유형 선택 695
WHENEVER SQLERROR CONTINUE문 19
WHENEVER문
오류 처리 135
SQLCA를 이용한 오류 표시기 18
SQL문과 함께 사용시 유의사항 19
Windows 등록 데이터베이스
OLE 자동 UDF 472
Windows 코드 페이지
지원되는 코드 페이지 566
DB2CODEPAGE 레지스트리 변수 566
WITH HOLD로 선언된 커서
X/Open XA 인터페이스 617
WITH OPTIONS절
참조된 컬럼 범위 정의 332
컬럼 옵션 정의 332

X

X/Open XA 인터페이스 615
다중 스레드 응용프로그램 618
단일 스레드 응용프로그램 618

X/Open XA 인터페이스 615 *(계속)*
저장점 204
트랜잭션 615
트랜잭션 처리의 특징 615
API 제한사항 618
CICS 환경 615
COMMIT 및 ROLLBACK 616
DISCONNECT 616
SET CONNECTION 616
SQL CONNECT 616
WITH HOLD로 선언된 커서 617
XA 환경 617
XASerialize 618

[특수 문자]

#ifdefs, C/C++ 언어 제한사항 684
#include 매크로, C/C++ 언어 제한사항 667
#line 매크로, C/C++ 언어 제한사항 667

IBM에 문의

기술적인 문제가 발생한 경우에는 DB2 고객 지원 센터에 문의하기 전에 *문제점 해결 안내서*에서 제안한 조치를 검토하고 실행해 보십시오. 이것은 DB2 고객 지원 부서로 하여금 사용자를 보다 더 잘 지원할 수 있도록 사용자가 모을 수 있는 정보를 제공합니다.

DB2 Universal Database 제품에 대한 정보나 주문은 그 지역의 IBM 영업 대표나 공인 IBM 소프트웨어 재판매업자에게 문의하십시오.

미국에 사시는 분은 다음 번호 중 하나를 선택하여 전화하십시오.

- 고객 지원을 받으려면, 1-800-237-5511.
- 사용가능한 서비스 옵션을 알려면, 1-888-426-4343.

제품 정보

미국에 사시는 분은 다음 번호 중 하나를 선택하여 전화하십시오.

- 제품 주문이나 일반 정보를 얻으려면, 1-800-IBM-CALL (1-800-426-2255)이나 1-800-3IBM-OS2 (1-800-342-6672).
- 책에 대한 주문은 1-800-879-2755.

http://www.ibm.com/software/data/

DB2 월드 와이드 웹 페이지에는 새로운 소식, 제품 설명, 교육 일정 등에 관한 현재의 DB2 정보를 제공합니다.

http://www.ibm.com/software/data/db2/library/

DB2 제품 및 서비스 기술 라이브러리는 빈도 높은 질문(FAQ), 수정사항(fixes), 책 및 최신 DB2 기술 정보에 대한 액세스를 제공합니다.

주: 이러한 정보는 영어로만 제공됩니다.

http://www.elink.ibmlink.ibm.com/pbl/pbl/

여기에서는 책을 웹 사이트에서 주문할 수 있는 방법을 제공합니다.

© Copyright IBM Corp. 1993, 2000

http://www.ibm.com/education/certify/
IBM 웹 사이트에서 기술 전문 인증 프로그램은 DB2를 포함하여 다른 IBM 제품의 기술 전문 인증 테스트 정보를 제공합니다.

ftp.software.ibm.com
anonymous로 로그인하십시오. `/ps/products/db2` 디렉토리에서, DB2와 많은 관련 제품에 관한 데이터, 수정사항, 도구 등을 찾을 수 있습니다.

comp.databases.ibm-db2, bit.listserv.db2-l
이러한 인터넷 뉴스 그룹으로 사용자는 DB2 제품에 대한 자신의 사용 경험을 토론할 수 있습니다.

Compuserve에서, GO IBMDB2
이 명령을 입력하여 IBM DB2 계열 포럼을 액세스하십시오. 모든 DB2 제품이 이러한 포럼을 통해 지원됩니다.

미국 외 지역에서 IBM에 연락하는 방법에 관한 정보는 *IBM Software Support Handbook*의 Appendix A를 참조하십시오. 이 문서에 액세스하려면, 웹 사이트 http://www.ibm.com/support/로 가서 페이지 맨 밑에 있는 IBM Software Support Handbook 링크를 클릭하십시오.

주: 일부 국가에서는 IBM 공인 딜러는 IBM 지원 센터 대신 해당 딜러 지원 부서에 연락해야 합니다.

IBM 한글 지원에 관한 설문

FAX : (02) 781-7778

보내 주시는 의견은 더 나은 고객 지원 체제를 위한 귀중한 자료가 됩니다.
독자 여러분의 좋은 의견을 기다립니다.

책 제목:IBM® DB2® Universal Database
응용프로그램 개발 안내서
버전 7

책 번호:SA30-0992-00

성 명		직위/담당업무	
회 사 명		부 서 명	
주 소			
전화번호		팩스번호	
전자우편 주소			
사용중인 시스템	☐ 중대형 서버	☐ UNIX 서버	☐ PC 및 PC 서버

1. IBM에서 제공하는 한글 책자와 영문 책자 중 어느 것을 더 좋아하십니까? 그 이유는 무엇입니까?
   ☐ 한글 책자 ☐ 영문 책자
   (이유:)

2. 본 책자와 해당 소프트웨어에서 사용된 한글 용어에 대한 귀하의 평가 점수는?
   ☐ 수 ☐ 우 ☐ 미 ☐ 양 ☐ 가

3. 본 책자와 해당 소프트웨어에서 번역 품질에 대한 귀하의 평가 점수는?
   ☐ 수 ☐ 우 ☐ 미 ☐ 양 ☐ 가

4. 본 책자의 인쇄 상태에 대한 귀하의 평가 점수는?
   ☐ 수 ☐ 우 ☐ 미 ☐ 양 ☐ 가

5. 한글 소프트웨어 및 책자가 지원되는 분야에 대해 귀하는 어떻게 생각하십니까?
   ☐ 한글 책자를 늘려야 함 ☐ 현재 수준으로 만족
   ☐ 그다지 필요성을 느끼지 않음

6. IBM은 인쇄물 형식(hardcopy)과 화면 형식(softcopy)의 두 종류로 책자를 제공합니다. 어느 형식을 더 좋아하십니까?
   ☐ 인쇄물 형식(hardcopy) ☐ 화면 형식(softcopy) ☐ 둘 다

IBM 한글 지원 서비스에 대해 기타 제안사항이 있으시면 적어주십시오.

☺ 설문에 답해 주셔서 감사합니다.
귀하의 의견은 저희에게 매우 소중한 것이며, 고객 여러분들께 보다 좋은 제품을 제공해 드리기 위해 최선을 다하겠습니다.

Printed in Australia

SA30-0992-00